Towards Correct and Efficient Program Execution in Decentralized Networks: Programming Languages, Semantics, and Resource Management

KARL PALMSKOG

Doctoral Thesis
Stockholm, Sweden 2014

TRITA-CSC-A 2014:16
ISSN-1653-5723
ISRN-KTH/CSC/A–14/16-SE
ISBN 978-91-7595-283-3

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framläggges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi fredagen den 24 oktober 2014 klockan 14.00 i F3, Kungl Tekniska högskolan, Lindstedtsvägen 26, Stockholm.

Tryck: E-print

Abstract

The Internet as of 2014 connects billions of devices, and is expected to connect tens of billions by 2020. To meet escalating requirements, networks must be scalable, easy to manage, and be able to efficiently execute programs and disseminate data. The prevailing use of centralized systems and control in, e.g., pools of computing resources, clouds, is problematic for scalability. A promising approach to management of large networks is decentralization, where independently acting network nodes communicate with their immediate neighbors to achieve desirable results at the global level.

The research in this thesis addresses three distinct but interrelated problems in the context of cloud computing, networks, and programs running in clouds. First, we show how implementation correctness of active objects can be achieved in decentralized networks using location independent routing. Second, we investigate the feasibility of decentralized adaptive resource allocation for active objects in such networks, with promising results. Third, we automate an initial step of a process for converting programs with thread-based concurrency using shared memory to programs with message passing concurrency, which can then run efficiently in clouds.

Specifically, starting from fragments of the distributed object modeling language ABS, we give network-oblivious descriptions of runtime behavior of programs, where the global state is a flat collection of objects and method calls. We then provide network-aware semantics, that place objects on network nodes connected point-to-point by asynchronous message passing channels. By relying on location independent routing, which maps object identifiers to next-hop neighbors at each node, inter-object messages can be delivered, regardless of object mobility among nodes. We establish that network-oblivious and network-aware behavior in static networks correspond in the sense of contextual equivalence. Using a network protocol reminiscent of a two-phase commit for controlled node shutdown, we extend the approach to dynamic networks without failures.

We investigate node-local procedures for object migration to meet requirements on balanced allocations of objects to nodes, that also attempt to minimize exchange of object-related messages between nodes. By relying on coin-flips biased on local and neighbor load to decide on migration, and heuristics to capture object communication patterns, we show that balanced allocations can be achieved that make headway towards minimizing communication and latency.

Our approach to execution of object-oriented programs in networks relies on message-passing concurrency. Mainstream programming languages generally use thread-based concurrency, which relies on control-centric primitives, such as locks, for synchronization. We present an algorithm for dynamic probabilistic inference of annotations for data-centric synchronization in threaded programs. By making collections of variables in classes accessed atomically explicit, these annotations can in turn suggest objects suitable for encapsulation as a unit of message-passing concurrency.

Sammanfattning

2014 års Internet sammankopplar miljarder enheter, och förväntas sammankoppla tiotals miljarder år 2020. För att möta eskalerande krav måste nätverk vara skalbara, enkla att underhålla, och effektivt exekvera program och disseminera data. Den nuvarande användningen av centraliserade system och kontrollmekanismer, t ex i pooler av beräkningsresurser, moln, är problematisk för skalbarhet. Ett lovande angreppssätt för att hantera storskaliga nätverk är decentralisering, där noder som agerar oberoende av varandra genom kommunikation med sina omedelbara grannar åstadkommer gynnsamma resultat på den globala nivån.

Forskningen i den här avhandlingen addresserar tre distinkta men relaterade problem i kontexten av molnsystem, nätverk och program som körs i moln. För det första visar vi hur implementationskorrekthet för aktiva objekt kan åstadkommas i decentraliserade nätverk med hjälp av platsoberoende routning. För det andra undersöker vi genomförbarheten i decentraliserad adaptiv resursallokering för aktiva objekt i sådana nätverk, med lovande resultat. För det tredje automatiserar vi ett initialt steg i en process för att konvertera program med trådbaserad samtidighet och delat minne till program med meddelandebaserad samtidighet, som då kan köras effektivt i moln.

Mer specifikt ger vi, med utgångspunkt i fragment av modelleringsspråket ABS baserat på distribuerade objekt, nätverksomedvetna beskrivningar av körningstidsbeteende för program där det globala tillståndet är en platt samling av objekt och metodanrop. Vi ger därefter nätverksmedvetna semantiker, där objekt placeras på nätverksnoder sammankopplade från punkt till punkt av asynkrona kanaler för meddelandetransmission. Genom att vid varje nod använda platsoberoende routning, som associerar objektidentifierare med grannoder som är nästa hopp, kan meddelanden mellan objekt levereras oavsett hur objekt rör sig mellan noder. Vi etablerar att nätverksomedvetet och nätverksmedvetet beteende i statiska nätverk stämmer överens enligt kontextuell ekvivalens. Genom att använda ett nätverksprotokoll som påminner om en tvåstegsförpliktelse, utökar vi vår ansats till felfria dynamiska nätverk.

Vi undersöker nodlokala procedurer för objektmigration för att möta krav på balanserade allokeringar av objekt till noder, som också försöker minimera utbyte av objektrelaterade meddelanden mellan noder. Genom att använda oss av slantsinglingar viktade efter lokal last och grannars last för att besluta om migration, och tumregler för att fånga kommunikationsmönster mellan objekt, visar vi att balanserade allokeringar, som gör framsteg mot att minimera kommunikation och tidsfördröjning, kan uppnås.

Vår ansats för exekvering av objektorienterade program i nätverk använder meddelandebaserad samtidighet. Vanligt förekommande programspråk använder sig generellt av trådbaserad samtidighet, som kräver kontrollcentrerade mekanismer, som lås, för synkronisering. Vi presenterar en algoritm som med dynamisk probabilistisk analys härleder annoteringar för datacentrerad synkronisering för trådade program. Genom att göra samlingar av variabler i klasser som läses och skrivs atomiskt explicita, kan sådana annoteringar antyda vilka objekt som är lämpliga att kapsla in som en enhet i meddelandebaserad samtidighet.

Acknowledgements

I would first like to thank my advisor, Mads Dam, for initially inspiring me to learn (and apply) formal methods in a course back in 2005, supervising my master's project, and providing guidance, advice, and feedback during my doctoral studies.

Special thanks are due to my coauthors, in particular Andreas Lundblad, Peter Dinges, and Alberto Gonzalez Prieto. Musard Balliu and Peter Dinges additionally provided valuable feedback on earlier drafts of this thesis.

Many senior colleagues have provided advice and enlightening discussions along the way. In particular, I am grateful to Dilian Gurov for encouragement, advice, and mentoring. I also thank Rolf Stadler, Stefan Arnborg, Stefan Nilsson, Gul Agha, Johan Boye, Douglas Wikström, Johan Håstad, Erik Fransén, Henrik Eriksson, Einar Broch Johnsen, and Rudolf Schlatte.

I thank all my coworkers at KTH, past and present, for providing a friendly environment conducive to research, studies, and teaching. In particular, I am grateful to Kristján Valur Jónsson, Torbjörn Granlund, Björn Terelius, Musard Balliu, Pedro de Carvalho Gomes, Gunnar Kreitz, Cenny Wenner, Linda Kann, Siavash Soleimanifard, Rebekah Cupitt, Emma Enström, Per Austrin, Lukáš Poláček, and Oliver Schwarz. I also thank my former colleagues at Jaycut AB, in particular Jonas Hombert and Gustav Broberg.

I am grateful to my friends outside academia for their support and patience, and for distracting me from work at reasonable intervals. Thank you, John-Philip Johansson, Stefan Svebeck, Steve Olofsson, Joakim Bennedich, Filip Marzuki, and Aron Petterson. I thank my family for support, most of all my parents, Kajsa and Göran Palmskog.

Stockholm, September 2014
Karl Palmskog

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In the early 1960s, J. C. R. Licklider, a psychologist and computer scientist at the Massachusetts Institute of Technology, described the concept of an Intergalactic Computer Network, in which geographically separate interconnected computers would provide access to computational resources to individuals, regardless of location, and facilitate information exchange [118, 165]. Since then, with the development of the Internet, expected to connect a number of devices upwards of 50 billion in 2020 [67], significant strides have arguably been made towards making that concept a reality. However, the growth of networking and networks has brought to the fore a wealth of problems spanning across computer engineering and theoretical computer science, among them (1) the problem of effectively scaling networks to handle more complex computations and larger amounts of data for processing and dissemination, (2) the problem of allocating resources in networks to meet performance objectives, and (3) the problem of using interconnected computers to execute programs correctly and predictably.

The research presented in this thesis follows a line of investigation that proposes *decentralization* as a promising way to achieve scalability and manageability of large-scale networks [122], while maintaining predictable and correct concurrent program execution on network nodes. The perspective is mainly from the viewpoint of formal methods and programming languages, and thus focuses on rigorous analysis using logic and algebra.

Overview This chapter is structured as follows. Section 1.1 gives the background of the research in the thesis, both in terms of the scientific literature and the associated real-world problems. Section 1.2 motivates the general approach. Section 1.3 describes the research problems in more detail. Section 1.4 gives a high-level view of several rigorous models of concurrency and distribution relevant to this thesis. Section 1.5 defines and motivates the choice of network model. Section 1.6 describes the specific problems addressed in this thesis, and Section 1.7 gives an overview of the papers found in later chapters. Section 1.8 concludes and covers future work.

1.1 Background

The research in this thesis is motivated by several trends and strands in the evolution of the Internet, networking and programming languages research.

1.1.1 The Internet: Past, Present, and Future

Development of large-scale, packet-switched data networks from the 1960s and onwards was motivated by both visions of universal access to computing resources and the potential applications in military communications [118, 91]. Early work on internetworking focused on robustness and survivability, in particular preserved communications in the face of significant losses of nodes in underlying networks [118]. Such properties were deemed desirable both for use in warfare conditions [175] and for providing remote access in face of inherent failures of network components and computers [91].

The multitude of networks comprising the Internet were assumed to be largely static over time, with addresses, as specified by the Internet Protocol (IP), corresponding to computers on specific sites (locations) [166]. Processes, in contrast, were described as the active computation element, executing on such host computers and communicating reliably one-on-one with other processes, whether near or remote, via the Transmission Control Protocol (TCP) running on top of IP [35].

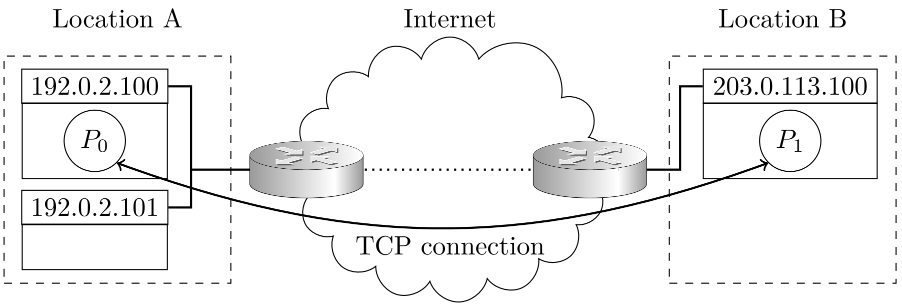


Figure 1.1: Illustration of internetworking

Figure 1.1 gives a conceptual view of internetworking. To the left, at Location A, there is one computer with IP address 192.0.2.100, where the process P_0 runs. At the same location, there is another computer with IP address 192.0.2.101, with a direct physical connection to the first computer; their proximity is reflected in the small address difference. To the right, at Location B, there is a single computer with IP address 203.0.113.100, where the process P_1 runs. The processes at the two sites have established a TCP connection between them, with data flowing in packets from one local network, via a gateway router, through a number of heterogenous networks, to the other local network.

The Open Systems Interconnection (OSI) model [99] standardizes and abstracts the functionality involved in networked communication into different layers, and was formulated in the beginning of the 1980s from the experiences with the early Internet and its progenitor ARPANET [210]. As with any standard, one purpose was to allow modularity and promote competition between implementors.

	#	**Name**
Application	7	Application
	6	Presentation
	5	Session
Data Transport	4	Transport
	3	Network
	2	Data Link
	1	Physical

Figure 1.2: The OSI model

Figure 1.2 gives a conceptual view of the model. The bottom layer (1) is the physical layer, which deals with (possibly unreliable) physical transmission of bits. Layer 2 is the data link layer, which concerns exchange of blocks (units) of data between named devices connected through physical links. Layer 3 is the network layer, which handles transmission of packets between hosts, possibly residing on separate physical networks (any network connected to the Internet). Layer 3 data transmission may therefore require inter-network routing based on host addresses. Layer 4 is the transport layer, which allows for ordered, lossless data (packet) transmission across hosts. This layer can provide an abstraction similar to a closed circuit between two parties to transmit information, as in telecommunications. Layer 5 is the session layer, that abstracts interhost communication, e.g., in terms of messages. Layer 6 is the presentation layer, that handles delivery and formatting of data for the topmost application layer (7). The layers can be partitioned into two sets, as shown in the figure, with the four lowest layers comprising the set concerned with data transport, and the three upper layers comprising the set related to applications.

A realization, or implementation, of a layer in a communication model such as OSI is a *network protocol*. An implementation of several contiguous layers (a suite of protocols) is usually referred to as a *stack*. Figure 1.3 illustrates communication between two stacks based on the OSI model, making the distinction between actual data flow, and data flow as it appears to layers above, the *logical* flow. In operating systems, a “network stack” usually refers to an implementation of the complete TCP/IP protocol suite, which notably collapses OSI layers 1 and 2 into a network access layer, and layers 5, 6, and 7 into an application layer. In this thesis, the network functionality and characteristics assumed corresponds to OSI layer 2, as described in more detail in Section 1.5.

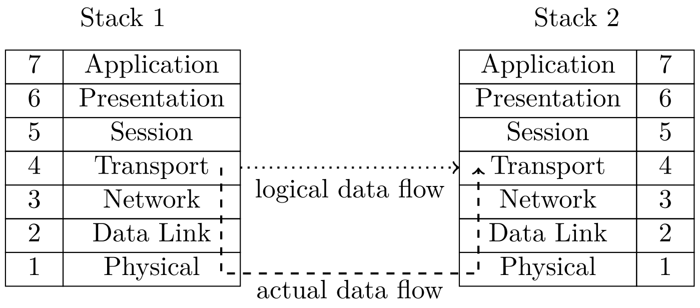


Figure 1.3: Two-party transport-layer communication in the OSI model

A number of assumptions underlying the design of the TCP/IP protocol suite have become less convincing over time, although the main abstraction it introduces for process-to-process communication—the network socket—is still widely used. For example, computers can be, and often are, mobile across locations, and in some systems it may be advantageous for processes themselves to be mobile across computers [28]. Furthermore, computer communication is increasingly transmission of *content*, such as video, from producers to consumers, which is more efficient if performed as a broadcast from one source to multiple receivers rather than as a number of (possibly concurrent) two-party socket-based sessions [113].

The Internet as of 2014 contains a multitude of workarounds for the deficiencies of TCP/IP, mostly implemented at the application level. For example, Content Distribution Networks (CDNs) build application-layer solutions for geographically-sensitive content retrieval by unreliably mapping IP addresses to approximate locations [170]. Other workarounds, such as Mobile IP [167], that provides basic support for mobile devices using IP, introduce some conservative protocol changes.

In the long run, it is preferable to address the fundamental architectural issues from a clean slate rather than maintaining increasingly complex workarounds that are backwards compatible, but the short-run costs to change architectural components can be large or require significant coordination among otherwise competing parties [10]. For example, IPv6, a conservative extension to the prevailing IPv4 protocol which solves the problem of a shrinking global reserve of 32-bit IP addresses, is as yet not widely used [49]. In addition, enormous amounts of resources have been invested in the current Internet infrastructure; hence, there is significant inertia in the adoption of new basic components and principles of networking, which has been referred to as Internet *ossification* [138].

Researchers have developed an abundance of solutions for perceived problems in internetworking. In Europe, the EU 7th Framework Programme project 4WARD [1] proposes an architecture of the future Internet, called Network of Information (NetInf) [55], that puts data itself at the center, rather than devices or locations; this concept was initially suggested under the name content-centric networking [100].

In the USA, the National Science Foundation has funded several projects related to a clean-slate future Internet, such as the Named Data Networking (NDN) project [149]. Similarly to NetInf, NDN proposes an architecture that enables content distribution without knowledge of devices or locations [209]. Many of these proposals involve radical changes to the boundaries of the network stack, and come with a host of new assumptions about services and software.

An important aspect of the future Internet is the struggle between centralized and distributed control. Commercial and governmental interests generally favor centralized systems and decision-making, because they can be more easily controlled and regulated. Besides the potential clash of interest with network users, this ease of management can be accompanied by higher susceptibility to failure, as shown by an incident that involved the hierarchical (and thus not fully decentralized) DNS system of name-to-address lookup that rerouted a significant part of Internet's traffic away from intended destinations [140, 202]. On the other hand, development of systems with distributed control, e.g., peer-to-peer systems, is not always motivated by a desire of performance or robustness, but the desire to avoid regulation or surveillance [5].

The research in this thesis uses decentralization for its technical merits, and can be viewed as an argument to design for decentralization by default, and only centralize data and control when absolutely necessary, similarly to the proposed *zero method* in social science research [171], in which rationality of actors is an assumption only deviated from when other options have been ruled out. A decentralized architecture using asynchronous message passing can ensure adequate performance in the face of unexpected deterioration of deployment conditions for a system, e.g., load increases or increased uncertainty on latency bounds.

1.1.2 The Cloud Computing Paradigm

The early Internet provided remote access to specific computers for researchers, with specific installed programs and data stored on the hard disks. However, most users are not interested in the computers themselves, but in the resulting computations and data (or, more accurately, the presentation of the resulting data through some interface). There is therefore no inherent reason for them to directly access and manipulate the computing resources they are using, beyond providing commands or programs as input. In fact, installing and maintaining computing infrastructure is a daunting task [41].

Cloud computing is a paradigm that has come to prominence in the last decade, where pools of computing resources are shared between applications accessed over a network, usually the Internet [102]. When a pool is provided by an organization to third parties (often at a cost related to usage), it is referred to as a *public cloud*, while a pool maintained for organization-internal use is a *private cloud*. For reasons of cost effectiveness and flexibility, the primary way to construct (public) clouds is by using commodity hardware [83].

From a resource management perspective [102], the roles of the actors in cloud computing can be divided into (1) Cloud Provider, (2) Cloud User, and (3) End User. The Cloud Provider manages data centers and system software to provide either an application directly to the End User, or interfaces for use by the Cloud User to provide applications to the End User. The Cloud Provider shoulders the responsibility of provisioning resources from its pool as they are needed by Cloud Users or End Users, as codified in Service Level Agreements (SLAs). Cloud Users, in turn, can provide SLAs to End Users of their applications. A common classification of public clouds, based on the types of interfaces provided by the Cloud Provider, is as Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS). The meaning of these terms is still somewhat in flux, since cloud computing abstractions are not yet completely standardized. Figure 1.4 illustrates the different roles in cloud computing and their interrelations, and shows the structure of Cloud Provider interfaces.

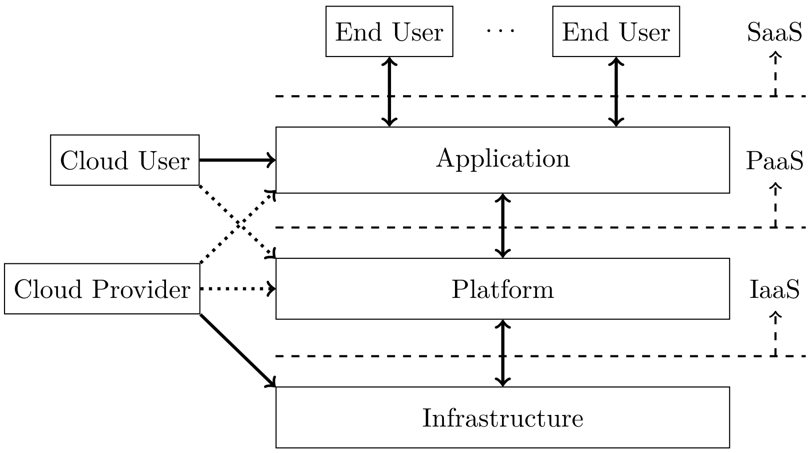


Figure 1.4: Roles and relationships in cloud computing

One example of a public PaaS cloud is Google App Engine [81], which lets Cloud Users run web applications through Google's infrastructure, with automatic resource provisioning (scaling) as traffic increases. Amazon Web Services (AWS) [8] is a de facto standard cloud computing platform that offers both IaaS and PaaS interfaces. The automatically scaling AWS PaaS for web applications is largely built on top of the IaaS interface, and is thus more flexible than Google App Engine.

While services such as AWS offer SLAs and extensive monitoring facilities for Cloud Users, the physical realization of processes and data in the datacenters is as a rule opaque; there is no way for a Cloud User to know whether an application behaves as expected except by black box testing or through the Cloud Provider's interface. Certain Cloud Users may require, for example, that data is never transported across country borders, or that data is never transported unencrypted through any network, but be unable to find out whether this actually occurs. Additionally, with an opaque cloud environment, there is no way for a Cloud User to make his own due diligence on whether a proposed SLA is fair given the price, except through historical data. With increased standardization and development of formal techniques for cloud management, Cloud Providers can provide iron-clad, transparent guarantees of correct execution of programs and show calculations behind SLAs backed by system design [199].

1.1.3 A Software-Hardware Divide in Flux

In a classic book on operating systems [194], Tanenbaum notes that, in the design of computer systems, whether certain components are realized physically in hardware or abstractly in software is often a question of tradeoffs between cost, performance, and flexibility that can be reconsidered over time. This observation is particularly pertinent in the design of networking systems, where expected network speeds have increased by at least three orders of magnitude over the last twenty years, all while requirements on, e.g., management and security have multiplied [146]. Traditionally, commercial vendors of networking forwarding devices, such as routers, have traded off flexibility for lower cost by producing Application Specific Integrated Circuits (ASICs) that incorporate the forwarding logic. This tradeoff becomes inadequate in environments where the forwarding logic must be changed frequently. The other extreme option is to run all the forwarding logic in software, e.g., on top of commodity hardware, in effect trading off performance for flexibility. Hybrid approaches are an active research area [146].

At a slightly higher level of abstraction, there has recently been attention towards Software-Defined Networking (SDN) [154], or programmable networks. In this approach, network forwarding logic is separated from control decisions, which are instead moved into software controllers that typically have access to more context-specific information. Software developers can thus tailor network behavior for specific applications and enforce policies as they would with other resources such as storage. Where previously many different network devices made separate and possibly uncoordinated decisions, in SDN they can be centralized.

The interest in cloud computing has motivated recent work on virtualization of operating system instances [16]. Historically, sharing of a physical computer between individuals has been done at the operating system level, so that the operating system maintains information on users, their credentials, and access rights, and enforced access policies for logical storage and devices. In addition, the operating system ensures, more or less successfully [172], separation of physical memory access and division of Central Processing Unit (CPU) time between different processes. Now, using support for virtualization in both operating systems and hardware, a physical computer can host many operating system instances, or virtual machines, each accessed by different users who usually remain oblivious to one another. With a verified hypervisor, which manages instances and checks system calls, such separation properties can be certified [50].

1.1.4 Towards Concurrency, Distribution, and Asynchrony

For a long time, programmers could assume that performance on sequential tasks roughly doubled every other year due to advances in computer hardware alone [147]. This is no longer the case, since clock rates of chips no longer increase at the required pace. Instead, the number of cores per chip has increased [80], and effective use of multiple cores for performance generally requires careful use of concurrency primitives in programs. Theoretically, performance is then constrained mainly by the program parts that must be run sequentially [9].

While the development of multi-core chips is motivated principally by the possibility of performance increases in existing programs through parallelism, there are many types of systems that incorporate simultaneity by nature, such as sensor networks [206]. One useful description of concurrency in such contexts is as nondeterministic composition of programs or their components [85]. Understood in this way, concurrency is ubiquitous in computing [142], both as the way to implement sequential programs and systems, and in its own right.

Mainstream operating systems and programming languages generally expose a thread-based model to programmers. In thread-based concurrency, a number of separate control flows (lightweight processes) run side-by-side in a shared-memory environment. Hence, all such threads can potentially access and modify the same global state. This model is the norm both in classical imperative programming languages such as C [196], and in object-oriented languages such as Java [164] and C++ [98]. Without the use of primitives that restrict concurrent behavior, running threads can interfere with each other, e.g., compete for reading and writing certain global variables, with unexpected results. Whether consciously formulated by programmers or not, many sets of variables in threaded programs have certain logical relationships between them, often called *invariants*; with unrestricted thread behavior, such invariants cannot reliably be upheld, making predictable program execution all but impossible. Concurrency primitives allow programmers to guarantee that certain sequences of operations in a thread are performed atomically, that is, without the possibility of interference from another thread. Still, conventional control-centric primitives, such as locks and monitors, are notoriously difficult to use; a real-world study of concurrency bugs in multithreaded programs suggests that fully half of all such bugs are atomicity violations [128].

Figure 1.5 shows an example of a how, due to timing of unconstrained read and write operations, one thread can end up with inconsistent information about a data structure in shared memory; the inconsistency can be described as being due to a (high-level) data race. In the scenario, there are two threads, A and B, and two shared data fields: a linked list, e.g., as represented by its first element and a next pointer, and an integer that is the size of the list. Initially, thread A writes a new value to the list field. However, before thread A updates the size field, thread B reads that field, which holds the value relevant for the old list. Thread B then reads the list field, containing the updated list, and thus ends up with inconsistent data. The situation could have been prevented if atomicity was separately enforced for both the writing operations of thread A, and the reading operations of thread B.

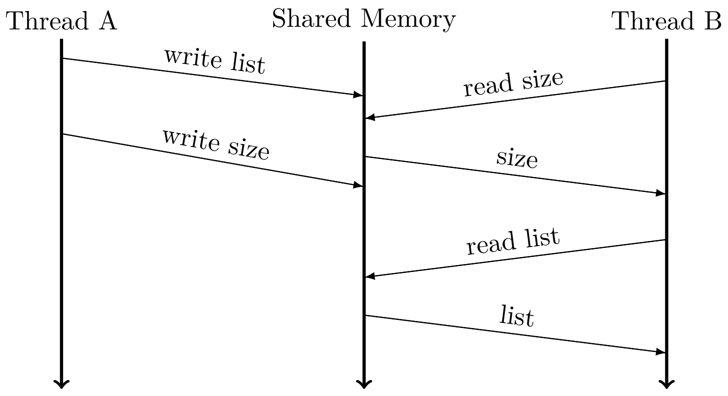


Figure 1.5: Data race causing violation of an invariant

The main alternative to thread-based concurrency is message-passing concurrency, where processes interact only by sending and receiving discrete messages through some medium (e.g., network wire, or memory buffers). This enables localized reasoning about the state of each process; interference freedom [156], which is necessary to provide guarantees about behavior in a threaded setting, holds trivially inside procedures executed in isolation. However, there is still the possibility of deadlocks, manifested as processes possibly waiting forever to receive messages that will not arrive, and livelocks, where processes are somehow active but unable to progress.

Systems that are concurrent by nature are often *distributed*: the components that are nondeterministically composed run on physically (or, given Section 1.1.3, logically) separate *sites* [86], as do networks. Famously, a distributed system can be defined as one in which an unknown computer located elsewhere can cause local failures [115]. In distributed systems, the message-passing model generally mirrors more closely what actually goes on, and is thus more straightforward to implement, although a shared memory can be simulated with some difficulty [120]. In fact, even at the single-chip level, modern processor architectures resemble message-passing systems.

A fact of life in distributed message-passing systems is asynchrony: the lax, or nonexistent, bounds on latency of communication between sites and processes residing there. With a low latency, processes can perform actions that appear as though they were handshakes to observers. One of the crucial properties of a mostly synchronous system is that conclusions can be drawn from the *absence* of some message or event. Mostly asynchronous systems cannot draw reliable conclusions from the timings of events—only through causality, i.e., the happens-before relation. If possible, designing systems to function correctly and efficiently in distributed, asynchronous environments is arguably to be preferred, since at the time of construction, runtime details such as latency or deployment sites may not be known, or be uncertain to a large extent. Algorithms designed for running in synchronous systems can divide computations into *rounds* to, e.g., ensure participation of every node. Asynchronous algorithms can simulate similar behavior using synchronizers [13], at the cost of being limited by the latency of the slowest component.

1.2 Motivation

The research in this thesis focuses directly or indirectly on what may be called an extreme case of networking: nodes without centralized control communicating via point-to-point asynchronous message passing using OSI layer 2 interconnects. Since fundamental problems such as consensus, i.e., agreement on some data value among nodes, cannot be solved in a fully asynchronous setting in the presence of faults [70], it is sometimes necessary to introduce some synchrony to achieve applicability to real-world settings. By way of operating system virtualization, virtual networks, and clouds, systems designed with decentralization and asynchrony in mind have wider applicability than simply running them in physical networks with these properties.

The strongest argument for decentralized networks is that resource control in such systems can be expected to scale to beyond in the order of thousands of nodes, the limit in centralized systems [102]. In addition, they can offer easier scaling; new nodes that are added do not need to registered and processed centrally, and nodes that are to be retired only need to finish communication with their neighboring nodes, if at all. The neighbors may be few or even constant relative to the size of the network, in particular for grid topologies, as demonstrated in Figure 1.13.

If self-managed decentralized networks are used as a foundation for building a public cloud, such as a PaaS, developers can be freed from the burden of resource allocation even outside of specific domains such as web applications. Cloud Users can also be isolated from management of various failures, related to, e.g., hardware, network congestion, and unresponsiveness. Ultimately, End Users can be provided with better services at lower cost than before.

Cloud Users require correct behavior of their programs when deployed in a PaaS. Correct means that the program behavior in the cloud in some way corresponds to the behavior the programmer expects after reading the program code and observes when running the program locally. This correspondence can be defined and reasoned about rigorously when (1) the semantics of the programming language is itself rigorously defined and (2) the cloud environment (consisting of nodes and links) is modelled precisely and accurately.

From the Cloud Provider's perspective, correctness is only the base requirement for an IaaS or PaaS cloud; resource management such as relocation of computations and data between nodes, and (de)activation of nodes needs to be done while maintaining the semantic correspondence. A commercial Cloud Provider, aiming to maximize enterprise value, will attempt to minimize resource usage costs over time while upholding SLAs.

The programming languages considered in this research for PaaS deployment use asynchronous message-passing as the fundamental operation, which is not the default in commonly used object-oriented languages. Hence, for Cloud Users to be able to use a PaaS, programs must be designed from scratch for message-passing semantics or be converted from the thread-based model. Because it is not reasonable to expect large-scale manual conversions of legacy programs in the latter case, there is a need for automatically inferring concurrency semantics, to repurpose programs for the message-passing model.

1.3 Research Problems

Consider a network of nodes with OSI link-layer interconnects. If all control, e.g., decision-making on resource allocation, is node-local, all coordination must be accomplished through message passing between neighboring nodes. The only way for nodes to obtain global information, such as the average load, is then through aggregation of information by coordinated message passing, e.g., by the formation of a tree overlay on the network graph and flow of partial aggregates along branches [54]. Generally, the problem is one of devising (and proving correct) distributed algorithms for efficient coordination and data aggregation that do not require significant synchrony or rely on centralized facilities.

Notoriously, without using problem-specific restrictions, proofs of properties in rigorous models of networked systems must consider all possible compositions of node events, which, without a bound on the number of nodes, is infinite. Without sound abstraction, this rules out techniques based on analysis of bounded state spaces, e.g., on-the-fly, or symbolic, model checking [39], except as an aid to debugging. One possible route is to formulate the system as a transition system in an inductive framework. This enables using rule induction to prove safety properties [86]. Bisimulation-like equivalences between systems, explained in Section 1.4, are more straightforwardly established through coinductive proof methods [179].

To have guarantees of program execution at runtime, proofs need to be done on the level of the language, rather than at some more abstract level that does not capture all implementation-relevant details. For example, the Java standard library implementation of a binary search algorithm on arrays, proven correct on mathematical integers, for a long time provided incorrect results for large inputs due to unforeseen machine integer overflows [23]. Executable programs proven functionally correct still generally rely on many assumptions about the runtime environment. To achieve greater reliability it can therefore be argued that all significant parts of the computing environment, such as networking hardware, computer hardware, and operating systems, should also have their functionality specified formally, and be proven correct with respect to specifications.

If the computational unit in a cloud is referred to as an *object*, the problem that faces the Cloud Provider may be called the *object placement problem.* More precisely, it is the problem of continually allocating these objects to nodes so that certain performance objectives are met (as specified in SLAs), possibly with the side condition of minimizing costs related to infrastructure. For an IaaS cloud, mobile objects would be operating system instances—virtual machines—rather than containers of processes and data as in the PaaS case. Note that the cost of mobility itself, in terms of lost throughput and increased latency, must also be taken into account. Figure 1.6 shows one possible solution to an object placement problem, with dotted lines indicating assignment of objects to nodes, and dashed lines between objects indicating the possibility of object-to-object communication. Note that while the assignment is balanced, in that no load is unnecessarily over- or under-loaded in terms of objects, it is not optimal with respect to minimizing communication, since all pairs of possibly communicating objects, e.g., o_4 and o_3, are located at different nodes.

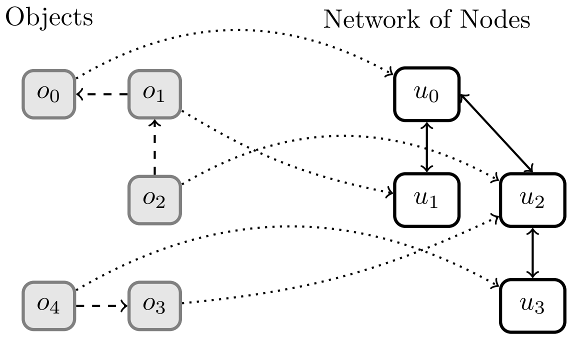


Figure 1.6: An object placement problem solution

Even if object placement can be addressed, solving the problem of *location transparency* among mobile objects is necessary to enable inter-object messaging, which is important many areas of distributed computing. When one object located at some node sends a message to another object, possibly located elsewhere, the message cannot simply be forwarded to some location to ensure delivery. Suppose each object can be assigned a globally unique indivisible identifier, i.e., an identifier that cannot be hierarchically decomposed as can IP addresses. The solution implied by content-centric networking is to have messages be explicitly addressed to the identifier of the recipient. The problem remains, however, of how to route these messages to their destinations in spite of mobility.

Figure 1.7 illustrates the problem of host-based addressing of messages via TCP/IP to implement location transparency. In Figure 1.7(a), a message from object o_0 at node u_0 to o_1 at u_1 has been dispatched with address u_1, after querying a location database, whose mappings are shown in the upper rectangle. The message is then transported across the network using TCP/IP, but in Figure 1.7(b), when the message arrives, o_1 has moved to some other node u_3—as reflected in the location database. It is possible to avoid or recover from this scenario in several ways, such as by forwarding the message to the new location, or using locking to ensure that an object does not move when a message to it has been dispatched. But the fact remains that, as soon as o_1 relocates, the message suddenly has the wrong address, and is being transmitted to the wrong destination. If the message is explicitly addressed to o_1 instead, this conceptual problem disappears, and no explicit "recovery" becomes necessary.

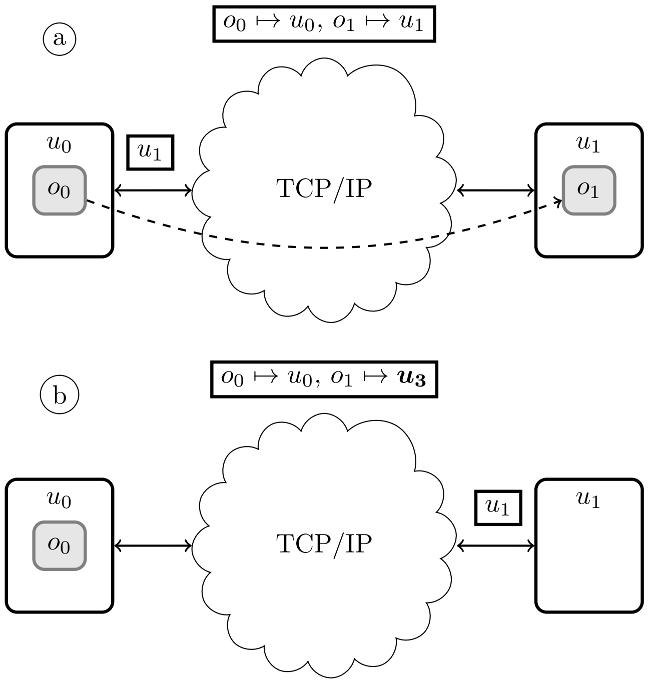

a
$o_0 \mapsto u_0,\ o_1 \mapsto u_1$
u_0
o_0
u_1
TCP/IP
u_1
o_1
b
$o_0 \mapsto u_0,\ o_1 \mapsto \boldsymbol{u_3}$
u_0
o_0
TCP/IP
u_1
u_1

Figure 1.7: Location transparency with host-addressed communication

In a particular network configuration, an inter-object message is traversing a specific path, or route, to reach its destination. The *stretch* of a route is defined as the ratio of its length, in terms of hops between nodes, to the length of the shortest possible path. A low stretch is desirable for routes in general; if a route is optimal, its stretch is one. Schemes for location transparency can thus be judged partly on their propensity to achieve low stretch, which is particularly difficult if they depend on hierarchical address formats for routing, as does IP.

The examples given of PaaS clouds with automatic scaling are specific to web applications, which have many restrictions that make implementation easier; web application frameworks such as Ruby on Rails [176] follow a *shared nothing* approach, where any state persistent between two web server requests is either due to the client (e.g., web browser cookies) or a separate database. Consequently, web server instances can be started or stopped at any time without regard to state (except possibly connection states of requests). The generalized problem of automatically scaling clouds must take node-local state into account when exercising power control over both virtual and physical instances.

1.4 Models of Concurrency and Distribution

The research in this thesis, while also of conceptual interest, mainly attempts to analyze network systems in terms of abstract, rigorous notions from logic and algebra. To rigorously describe sequential computational behavior, there are several established and widely accepted models, the two most prominent of which are the Turing machine [197] and the λ-calculus [38]. The basic element in the former is the reading or writing of data on a storage medium; in the latter, it is function invocation with actual parameters. None of these activities correspond well to what goes on in networks, where information exchange, or *interaction*, is central. Interaction also implies that several computational processes are active concurrently, usually on computers at different locations. In this setting, pure computation without interaction becomes a special case of the behavior of distributed processes.

Interaction between two parties in the physical world can be of at least two distinct kinds: a "handshake" that requires simultaneous (synchronous) action from both, or a signal (message) being transmitted from one to the other without the requirement of simultaneity. In providing a fundamental, rigorous description of interactive behavior, the decision whether to make handshakes or signaling primitive has fundamental consequences.

The research in this thesis brings together elements from both the π-calculus [144], which builds on synchronous handshakes, and the Actor model [3], which builds on asynchronous message-passing. Both of these formalisms have well-studied, rigorous foundations [181, 4] and have a wide variety of applications in both theory and practice [168, 177, 6, 65]. The main programming language used in the presented research, ABS, has a semantics that is close to the Actor model, but is here analyzed mainly using techniques that were established and popularized in the π-calculus tradition.

1.4.1 Process Algebras and π-calculus

Investigations into the foundations of concurrency beginning in the 1970s yielded several rigorous accounts of describing and reasoning about concurrent processes, notably CCS [141] and CSP [94]. In both CCS and CSP, processes, defined equationally, can perform labeled actions, possibly chosen nondeterministically, and be composed with other processes, with which they can interact. The meaning of a process expression is given in terms of a *Labeled Transition System* (LTS), which is a tuple of consisting of a set of states $\mathcal{S}$, a set of actions Act, and a transition relation $\mathcal{T} \subseteq \mathcal{S} \times Act \times \mathcal{S}$. An *execution* of a process is a possibly infinite sequence

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} s_n \xrightarrow{\alpha_n} \cdots$$

where $s_i \in \mathcal{S}$ and $(s_i, \alpha_i, s_{i+1}) \in \mathcal{T}$ for $i \geq 0$. The *trace* of such an execution is the sequence $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}, \alpha_n, \ldots$ of actions.

Figure 1.8 shows three processes represented as transition systems, with circles as states and labeled arrows defining the transition relation. Figure 1.8(a) and Figure 1.8(b) are classic systems [143] intended to abstractly capture the behavior of simple beverage vending machines. The machines provide either coffee or tea after the proper number of coins are deposited: one coin for tea, and two coins for coffee. It is easily seen that $\mathrm{coin}, \mathrm{coin}, \overline{\mathrm{coffee}}, \mathrm{coin}, \mathrm{coin}, \overline{\mathrm{coffee}}, \ldots$ is a valid trace of both systems. In fact, the possible traces are the same.

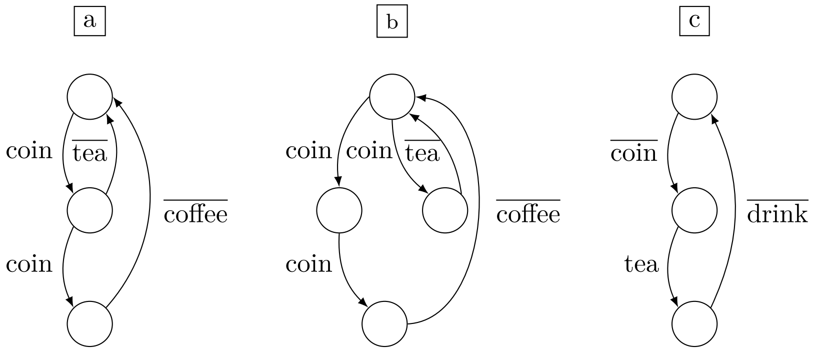

a
b
c
coin
$\overline{\mathrm{tea}}$
coin
$\overline{\mathrm{coffee}}$
coin
coin
$\overline{\mathrm{tea}}$
coin
$\overline{\mathrm{coffee}}$
$\overline{\mathrm{coin}}$
tea
$\overline{\mathrm{drink}}$

Figure 1.8: Labeled transition systems

Figure 1.8(c), represents the behavior of a compulsive tea drinker of limitless wealth, who repeatedly inputs one coin, gets a cup of tea, and then drinks it. Consider what happens when the tea drinker is composed in parallel with each vending machine, in turn, and starts to interact with them. In CCS, interaction is defined as two processes performing a mutual, synchronous transition on actions with complementary labels, in this case either coin and $\overline{\mathrm{coin}}$, or tea and $\overline{\mathrm{tea}}$. When interaction takes place, the result is a silent action, τ, which is not observable in a trace. Hence, in both composed systems, the only observable action is $\overline{\mathrm{drink}}$. Yet, to the tea drinker, the machines behave very differently; the first is a joy to use, but the second sometimes gets stuck, unable to provide him with the required tea. In these situations, the only way to progress, he finds, is to locate a coffee drinker and have that person deposit a single coin and retrieve the resulting cup of coffee.

π-calculus [144] is a more recent formalism, descended from CCS, which also relies on synchronous interaction as its primitive notion. π-calculus allows representing communication links and transmission of information among processes by primitive *names*. Since names are simultaneously channels and data, mobility, in the form of mobility of links in the virtual space of linked processes, becomes possible to describe and analyze. Among processes, access to channels can be controlled through the use of *name binding*, similar to variable binding in λ-calculus and algebra in general.

In the context of this thesis, the most important aspect of CCS and π-calculus are their notions of behavioral equalities between processes, and the methods for proving them. While equality is not a trivial notion is sequential computation, it can be straightforwardly understood through extensionality—equivalence by comparison of external properties. Regardless of how they are defined, two mathematical functions can be considered equal if they always map the same input value to the same output value. Two finite-state automatons, e.g., in the form of transition systems as in Figure 1.8, can be considered equal if they produce the same traces. Nondeterminism in a finite-state automaton, e.g., the automaton in Figure 1.8(b), does not fundamentally affect externally observable traces, since nondeterministic action can be reduced to the deterministic case [173].

The situation is different after introducing concurrency and interaction. To the tea drinker, there is a clear way in which the LTS in Figure 1.8(a) behaves differently from the LTS in Figure 1.8(b); for the former, it is always possible to get tea after depositing a coin, while for the latter, it is not possible if the machine is in the leftmost state. There is a distinct way in which the behavior of the first machine and the second machine are *distinguishable* by an agent interacting with both—if the second machine ends up in the leftmost state, it is unable to match, or *simulate*, the capabilities of the first machine. If an interacting agent is unable to find any such differences between two processes, they are said to be *bisimilar* [179], i.e., the first simulates the second and the second simulates the first.

One way of understanding bisimulation is as a game played between two processes, P_0 and P_1, that perform labeled actions. Suppose P_0 and P_1 are bisimilar, written $P_0 \sim P_1$, and P_0 performs an action, say α, and then evolves into the process P_0'. P_1 is then also able to perform α and then evolve into P_1', such that P_0' and P_1' are bisimilar. The converse case, when P_1 performs α, also holds. Figure 1.9(a) gives a diagrammatic presentation of the property, referred to in the literature as strong bisimilarity [143]. If, instead, a process is allowed to make an arbitrary, but finite, number of unobservable transitions τ before matching the action of the other process, the resulting equivalence is weak bisimilarity, written $P_0 \approx P_1$. Figure 1.9(b) shows a corresponding diagram, writing $P_1 \overset{\alpha}{\Longrightarrow} P_1'$ for $P_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_1'' \xrightarrow{\alpha} P_1'$.

1.4.2 The Actor Model and Active Objects

The Actor model was originally developed in the context of artificial intelligence research [92], but is now relevant as a foundation for designing, reasoning about, and implementing concurrent and distributed systems in general [3]. An actor has a unique name, independent control, and local state, and is able to send and receive messages from other actors. Messages received by an actor are placed in a (theoretically unbounded) private buffer, usually called a *mailbox*, until they are

$$\begin{tikzcd}
\textcircled{a} & & \textcircled{b} & \\
P_0 \arrow[r, phantom, "\sim"] \arrow[d, "\alpha"] & P_1 \arrow[d, "\alpha"] & P_0 \arrow[r, phantom, "\approx"] \arrow[d, "\alpha"] & P_1 \arrow[d, Rightarrow, "\alpha"] \\
P_0' \arrow[r, phantom, "\sim"] & P_1' & P_0' \arrow[r, phantom, "\approx"] & P_1'
\end{tikzcd}$$

Figure 1.9: Strong and weak bisimilarity

processed; hence, messaging is inherently asynchronous. In response to messages, actors can change their behavior, and thus their internal state, and spawn new actors. Figure 1.10 gives a conceptual view of an actor system.

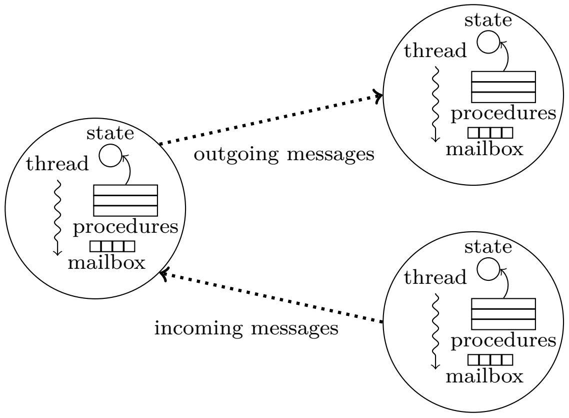


Figure 1.10: A system of actors

In an object-oriented programming language, the behavior of an object in response to a method call is determined by the definition of the corresponding method in the class of the object. The Actor model itself does not prescribe that behavior in response to a message is determined by the method name in this way. Instead, behavior can be determined by, e.g., pattern matching on the message, as in the Erlang programming language [65]. The combination of object-oriented programming language conventions, such as classes and methods, with the Actor model, is usually referred to as active objects [30, 31], although the difference from actors is not precisely defined in the literature [183].

Four key semantic properties of Actor systems [108] are (1) encapsulation of local state and messages, (2) fair scheduling, (3) location transparency, and (4) mobility. Encapsulation requires that an actor is unable to directly affect another actor's state—ruling out data races—and that messages contents are sent by value (immutable) and not by reference. Fair scheduling requires that messages are always delivered to their destinations, unless the recipient is disabled; this is taken to imply that no actor is permanently starved. Location transparency, as explained above, puts the burden on the system to separate locations from message destinations, even when locations can change as a result of mobility. In a survey of Actor frameworks for the JVM platform [108], however, only two out of seven frameworks guaranteed all of the key properties. In addition, programmers in practice tend to mix message-passing and control-centric concurrency primitives, e.g., locks, when using frameworks that permit this [195]. Still, the Actor model has had more impact in practice than process algebras and π-calculus, with industry-used languages and frameworks such as Erlang and its OTP libraries [66], and Akka on the JVM platform [6].

Futures [32, 72, 124, 207, 153, 56] are an abstraction for handling return values in active objects. A future, generated when a method is called asynchronously, can be viewed as a placeholder for the return value that may eventually be computed by the recipient object (the *callee*). When there is no value corresponding to the placeholder, the future is *unresolved*. After the destination object has received the message and computed the return value, the future becomes *resolved*, and stays that way forever. An object with access to a future f can attempt to claim the value associated with f, which could then result in blocking (busy waiting) or retrieval of the value, if it is available. Figure 1.11 illustrates how futures work in a system of active objects. Initially, object A, to the left, call a method in object B, thereby both dispatching a method call and creating a new future. When object B has received the method call and computed the resulting value, it updates the future with this value. Object A then gets the value of the future, possibly after doing other meaningful work.

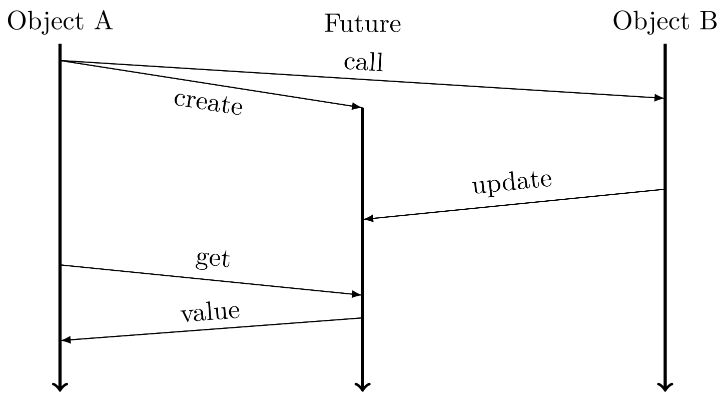


Figure 1.11: Sequence chart illustrating future creation and updating

1.4.3 The ABS Language

Most of the research in this thesis is related to the Abstract Behavioral Specification (ABS) language [103], developed in the EU FP7 HATS project. ABS is designed for expressing executable models of distributed object-oriented systems, and is related to the Creol language [104] and the CoBox model of concurrent objects [183]. Since ABS uses asynchronous message passing for method calls, and rules out data races, it can also be viewed as an actor language. More accurately, since actors do not use classes and methods, ABS can be called a language of active objects.

The canonical calculus of ABS, called Core ABS, consists of two distinct parts, or levels. First, it has a functional level of algebraic data types and terms, and enables defining side-effect free functions on such data. Second, above the functional level, is the object level, in which object interfaces and classes are defined, and which allows the concurrency semantics to be expressed.

The functional level code fragment in Listing 1.1 defines a parameterized binary tree data type `Tree` and a function `contains`, which, given a term `t` of type `Tree` and a term `a` of the abstract type `A` of tree elements, returns `True` if there is a such an element in `t`, and `False` otherwise. For instance, if `t` is the ground term `Node(1, Node(2, Tip, Tip), Node(3, Tip, Tip))`, then `t` has type `Tree<Int>`, and the expression `contains(t, 3)` is well-typed and reduces to `True`.

```
data Tree<A> = Tip | Node(A, Tree<A>, Tree<A>);

def Bool contains<A>(Tree<A> t, A a) =
  case t {
      Tip => False;
      Node(a, _, _) => True;
      Node(_, xt, yt) => contains(xt, a) || contains(yt, a);
  };
```

Listing 1.1: ABS functional level data type and function definitions

The code fragment in Listing 1.2 shows an example Core ABS interface and two implementing classes. The classes allow a programmer to construct a binary tree of objects, with each object holding an integer value `val`. If the method `aggregate()` is called on the root object in such a tree, the result is a cascade of method invocations flowing out towards the tree leaves, resulting in the summation of all values of the tree nodes. Note that, due to the initial non-blocking use of futures, the method calls of a non-leaf object to its two successors can be made concurrently.

ABS abstracts from many implementation-level concerns in distributed systems, which makes it more suitable for foundational investigations, at the cost of some relevance to actual implementations. One such abstraction, the use of interface types for objects, and by extension the mandatory hiding of implementation details for objects, is arguably necessary for distributed execution. ABS uses asynchronous method calls which return futures that can be claimed when values are needed. The

```
interface CastNode {      class LeafCastNode(Int val) implements CastNode {
    Int aggregate();          Int aggregate() { return val; }
}                         }

class BranchCastNode(Int val, CastNode left, CastNode right)
  implements CastNode {
   Int aggregate() {
      Fut<Int> fLeft = left!aggregate();
      Fut<Int> fRight = right!aggregate();
      Int aggregateLeft = fLeft.get;
      Int aggregateRight = fRight.get;
      return val + aggregateLeft + aggregateRight;
    }
}
```

Listing 1.2: ABS class and interfaces

semantics implements synchronous inter-object method calls by blocking the sender until the future is resolved.

The functional and object levels are accompanied by a type system, and a standard runtime semantics of Core ABS that can be described informally in terms of an evolving "soup" of objects with tasks and method calls in the form of messages. Specifying runtime behavior of objects in this way is influenced by Berry and Boudol's Chemical Abstract Machine [18] and the rewriting logic style of modeling distributed systems [40]. For well-typed programs, the semantics guarantees that certain undesirable events cannot happen, such as attempted invocation of nonexistent methods in the class of an object.

Figure 1.12 shows conceptual representations of three Core ABS runtime configurations, corresponding to snapshots of the same executing program. In Figure 1.12(a), there are two objects, o_0 and o_1, where the former knows the identifier of the latter. In addition, there is a resolved future f_0 whose value is accessible to both objects. In Figure 1.12(b), o_0 has made a call to a method in o_1, resulting in the creation of an unresolved future f_1 and a message addressed to o_1. In Figure 1.12(c), o_1 has received the message, computed the return value, and resolved f_1, as indicated by the changed container color.

The formal definition of Core ABS used in the research presented here is given in Appendix A; in comparison to the original definition [103], a number of ambiguities and minor errors have been fixed, and the unit of concurrency is a single object rather than a group of objects (concurrent object group, or cog). The latter change is motivated by making object migration between network nodes conceptually simpler, and is not fundamental for the research results.

The combination of a separate category of expressions, which are evaluated without side effects, and a semantics where all access to objects is through interfaces, makes it easier to implement a language in a distributed setting. Specifically, inter-object messages only need to contain as arguments immutable ground terms (fully

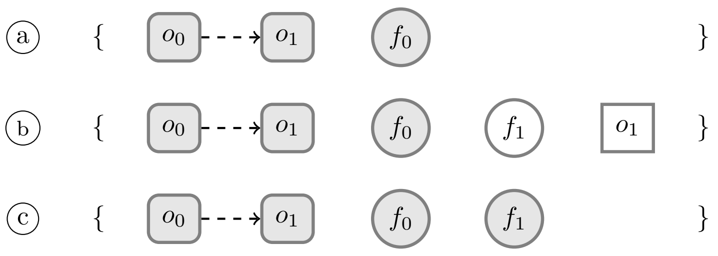


Figure 1.12: ABS runtime configurations

evaluated expressions), future identifiers, and object identifiers typed by interfaces. In a language such as Java, where expression evaluation can have side effects, and knowledge of the implementing class may be necessary to access an object, it is problematic to simply serialize arguments in method calls for network transport—the meaning of an argument can be different at the site of the receiver.

1.5 A Network Model With OSI Layer 2 Interconnects

The research in this thesis assumes a network model corresponding to OSI layer 2: nodes interconnected point-to-point by asynchronous message channels. Each node thus has a unique name (or identifier) and is aware of a number of neighboring nodes with which it can communicate by dispatching messages through the appropriate channel, with no chance of loss (unless the receiver shuts down). At any time, a message may be available through an incoming channel, but there is no requirement to process available messages immediately; hence, a node cannot know when a sent message has been received—only indirectly through a message received in return. Underlying layers are assumed to perform error correction so that dispatched messages are never lost. Nodes are assumed to act cooperatively and honestly. The model does not consider resources, e.g., in terms of CPU cores or memory, explicitly. In general, network messaging is considered much more expensive than local computation.

In more rigorous terms, a network is a *graph* with nodes as *vertices* and channels as directional *edges*. The graph is symmetric, i.e., whenever some vertex u has an edge to another vertex u', there is a corresponding edge from u' to u. The graph is also reflexive in that all vertices have an edge to themselves (a self-loop channel). Finally, the graph is connected, so that there is always a path between all pairs of vertices. Figure 1.13 gives a pictorial representation of this kind of network graph, in this case one which has a grid topology.

Interpreting a node, or a site, as a *possible world* in the sense of modal logic, the model corresponds to S5, where the accessibility relation between worlds is an equivalence relation [86]. For messaging, each edge in the graph is associated with an unbounded First In First Out (FIFO) message queue. Vertices can be associated

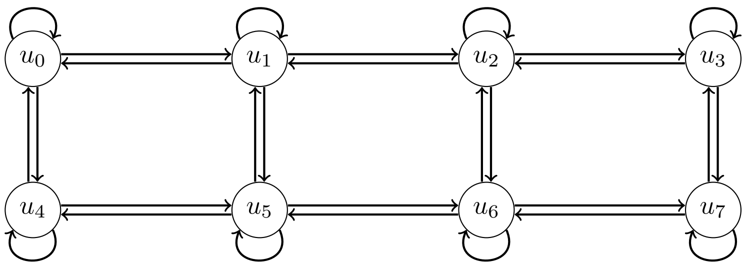


Figure 1.13: Network graph with grid topology

with local state, which can be related to programs executing in the network (e.g., objects) or management information (e.g., routing). The graph can be realized in several ways, most straightforwardly as a physical non-IP network, or a TCP-based overlay on an IP-based network.

A network corresponding to OSI layer 3 supports direct messaging between all nodes, commonly implemented using address-based routing, as for the Internet itself. The corresponding rigorous model is then most closely a complete graph, where all pairs of vertices have edges between them. OSI layer 4 ensures that, in addition, an abstraction resembling a closed circuit can be used for communication, i.e., communication order is preserved and error-checking and retransmission is performed when needed. In a decentralized system, where *objects* (either active objects or VMs) are the destinations of messages, there is no inherent reason for a node to communicate directly with non-neighboring nodes. As long as the network topology corresponds reasonably well to the physical layout, forwarding of messages to objects by nodes can also be done at least as effectively as when using OSI layer 3 functionality, and possibly with even less latency, when schemes with low stretch are used. There is, in addition, no reason for communication to resemble a closed circuit, since the aim is not communication between nodes themselves, but between objects. As long as communication is non-lossy at the link level, which is arguably a problematic assumption in realistic settings, the retransmission functionality of layer 4 is not needed.

1.6 This Thesis: Problems Addressed

1.6.1 Implementation Correctness of Active Objects in Decentralized Networks

In the research in this thesis, we consider several fragments the ABS language and its network-oblivious runtime semantics and propose several *network-aware* runtime semantics that take aspects of location, routing, and message-passing into account, i.e., describe execution at a level of abstraction close to implementation in a network. The network-aware semantics incorporate the network model described in Section 1.5 and use *location independent routing* for realizing location transparency for distributed objects. We consider relatively simple routing schemes where each node maintains a routing table that maps message destinations, which are object identifiers, to the identifiers of neighboring nodes which are the *next hops* to reach the destination (as far as is known locally). A central property of the set of reduction rules in all the network-aware semantics is that they are *local*, in the sense that each rule, corresponding to an atomic action in a distributed system, only uses node-local entities, e.g., links, routing tables, and local objects.

Figure 1.14 shows a fragment of a network-aware execution. In Figure 1.14(a), a message from o_0 to o_1 has been routed by the node u_2 towards u_1, according to the next hop entry in its local routing table, shown as the mapping $o_1 \mapsto u_1$. In Figure 1.14(b), o_1 has successfully migrated to u_3, with the routing tables at u_1 and u_3 updated as a result (although the routing table at u_0 remains stale). Consequently, the message is being routed using the new information towards u_3.

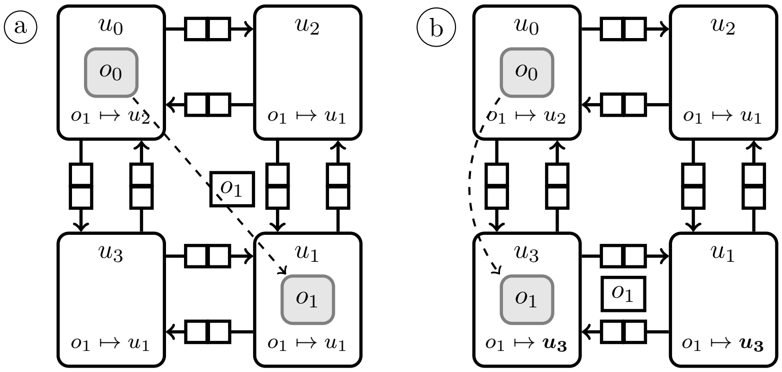


Figure 1.14: Network-aware program execution using location independent routing

With the precise formulation of the two program execution modes, it becomes possible to express whether program behavior in the network-aware semantics in some sense corresponds to the abstract behavior in the network-oblivious semantics. The notion of equivalence between runtime configurations in the two semantics used in this research is *contextual equivalence* [169], previously studied in the context of the π-calculus family of process formalisms, and related to weak bisimilarity, described in Section 1.4.1. Ultimately, Paper I and Paper II, summarized below, establish equivalences of this kind between network-oblivious configurations and their networked counterparts for two different ABS fragments.

The equivalence proofs rely critically on the possibility, at any point in an execution, of the information in routing tables becoming stabilized, i.e., sound and complete, if objects stay in their places. This makes it possible to deliver all outstanding object-addressed messages to their destinations, and, by extension, enables the formulation of *normal forms* for network-aware configurations that are close to the form of network-oblivious configurations.

Although the equivalence proofs are done at the language level, referencing specific ABS constructs, the techniques used are general and apply to similar languages and more sophisticated routing schemes. If convenient, a hierarchy of network-aware semantics at increasing levels of detail can be formulated, to come closer to actual implementation code. An advantage of this type of implementation correctness argument is the simplicity when compared to verifying the complete network stack, which was not constructed with formal analysis in mind [20].

In summary, the research contributes to theory and practice of implementing a network-based decentralized runtime system to support distributed, communicating, mobile active objects. In particular, the results provide arguments in favor of realizing location transparency by routing messages directly on object identities (actor names), and by extension, against relying on the transport or network layer in the current (TCP/IP) network stack.

1.6.2 Adaptive Resource Allocation for Distributed Objects in Decentralized Networks

A network-aware programming language semantics with unconstrained nondeterminism that, according to a suitable notion of contextual equivalence, corresponds to a network-oblivious semantics, effectively describes the state space available for object migrations in a given (static) network. But most “allowed” executions in a networked setting are nonsensical when viewed from a resource allocation standpoint; for example, objects can migrate to already-overloaded nodes, and objects can migrate forever without their tasks progressing. For a Cloud Provider, executions must fulfill certain performance objectives, e.g., evenly balanced load among network nodes. The problem is then to determine node-local behavior that accomplishes these objectives. An advantage to this approach is that objectives are ensured by the runtime system itself, rather than some separate, centralized resource allocator.

In existing clouds, resource allocation is typically performed in a centralized fashion [102], and may be considerably specialized for certain domains, such as for running web applications [81, 8]. An alternative approach, considered in Paper III, which is summarized below, is to run decentralized algorithms for achieving load balancing and other resource objectives. The algorithm in effect determines the *scheduling* of transitions in a network-aware semantics for a language of active objects. We study a measure of node load defined at the language level: the number of active computational tasks related to objects located at the node. In the Core ABS semantics, an object has at most one active task. Consequently, in the network-aware semantics for Core ABS, the node load is the number of objects with an active task. For load balancing, a key issue is to avoid oscillation, i.e., alternating, unstable allocations of resources over time. One promising technique to avoid oscillation is by introducing decision making based on results of coin flips [71].

The main probabilistic algorithm we evaluate for load balancing of distributed objects assumes a complete network graph for convergence to an evenly balanced allocation in expected polynomial time in the number of nodes. Such high connectivity is far from realistic in physical networks with link-layer interconnects. In addition, when the topology of a network is sparse, it becomes important to consider messaging between objects when migrating objects. If two objects located many hops away from each other exchange messages often, the result is significant network traffic and high latency of messages in terms of number of hops. It is thus important to consider at least two other measures besides node load, namely, link load (the number of messages passing through a link per unit of time) and message latency (the number of hops messages need to be routed).

Figure 1.15 shows a configuration that evolves to an ideal allocation through migration. In Figure 1.15(a), messages between, on one hand, o_1 and o_0, and on the other hand, o_3 and o_4, must be transported across the network. Additionally, node u_1 is overloaded with objects, and both u_0 and u_2 are underloaded. In Figure 1.15(b), all object messaging can be taken care of locally, and no single object migration can be performed which makes load more evenly balanced.

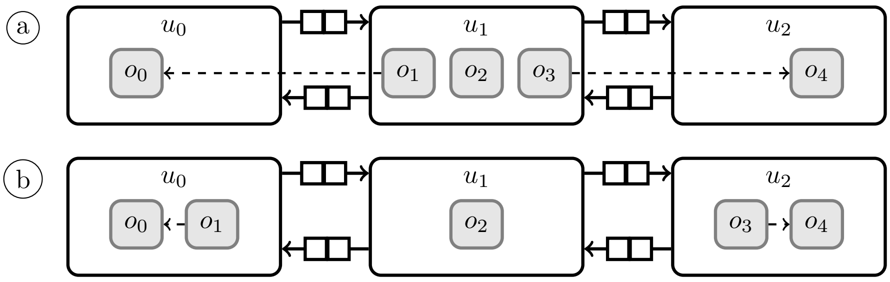


Figure 1.15: Object migration for load balancing and minimizing network traffic

A fundamental assumption in our resource allocation approach is that all decisions are made at runtime, rather than at development time (in the form of, e.g., annotations [105]) or at compilation time. Hence, no information about program characteristics is derived statically.

Our approach is to run the main load balancing algorithm, based on coin-flipping, in static networks with grid, hypercube, or full mesh topologies. We prioritize load balancing over other objectives. To achieve progress towards minimizing link load and message latency, we apply various heuristics when selecting objects to migrate and the destination neighboring nodes.

In Paper IV, also summarized below, we consider the case of benignly dynamic networks with asynchronous message passing, where there is no unexpected shutdown (or addition) of nodes, only due to stimuli from, e.g., the environment. Because crashes are ruled out, it is then not necessary to use replication, which can be a costly measure [136]. Instead, however, there is a requirement to ensure that no object-related message or object that is located on or inbound to a node are lost, when that node is shutting down. Only a node that has no objects, and whose outgoing and ingoing queues to all neighboring nodes contain no object or object-related message, can safely disappear without consequences to program execution—assuming the remaining network is connected.

In a decentralized asynchronous setting, a node that aims to shut down thus has to obtain agreement from its neighbors to ensure that no new objects or object-addressed messages are sent to it during the phase when all objects are migrated elsewhere. This process has the characteristics of a distributed transaction, or a *two-phase commit* [82, 117, 187]. Besides existing neighboring nodes, new adjacent nodes may also become activated as a node is shutting down, which could cause unwanted messages to be inbound from the new node. To avoid such situations, newly added nodes must not send object-related messages to a neighbor without agreement. As in a static network, the ultimate aim is to provide behavior that cannot be distinguished from the network-oblivious behavior, when "invisible" actions such as messaging related to node shutdown are disregarded.

From a resource allocation perspective, even if nodes can be triggered to safely shut down, there is still the problem of determining which nodes should be shut down at what time to simultaneously (1) maintain connectivity and (2) meet performance and power-related objectives, such as to minimize the number of active nodes for a given average load, when variance is within a certain interval.

There are limits to what can be achieved in practice with only asynchronous message passing, not least because of fundamental issues, such as the impossibility of consensus in the presence of failures. Nevertheless, many problems can be tackled to some degree, which raises the question of quantifiable tradeoffs between the amount of synchrony and how accurately problems in distributed computing, e.g., distributed aggregation and assignment of computational tasks to nodes, can be solved.

1.6.3 Program Conversion From Thread-Based to Message-Passing Concurrency

Concurrent programs written in mainstream languages, such as Java and C++, as a rule use thread-based concurrency primitives. As argued in Section 1.1.4, threaded code frequently contains atomicity bugs, and the assumption of shared memory is anathema to large-scale distributed systems, preventing execution in a PaaS cloud. Conversion of legacy programs to the message-passing model can thus lead to many benefits, most prominently the possibility of execution in a decentralized, scalable cloud environment. This may not be necessary or beneficial for all programs, e.g., those with highly optimized, fine-grained threading and synchronization. Yet, even when constructs for message-passing are available, many programmers still use, e.g., locks, because of familiarity and other non-fundamental reasons [195].

To properly convert an object-oriented program with locking to use message-passing, the concurrency semantics of the program must be understood and preserved. The concurrency semantics can be described, e.g., in terms of the control flows of multiple threads and how locking is used to guarantee atomicity in certain sections of the program—a difficult task. A more promising route is to consider concurrency semantics in terms of properties that hold between fields inside, and across, objects. In formal verification of sequential object-oriented programs, a common way to organize behavioral specifications is as *class invariants*, namely, as properties which hold at entrance and exit for all class methods, and at the exit of the constructor [133]. While weak class invariants, e.g., that some class field is never `null` or the null pointer, are feasible to formulate and establish for large programs, stronger invariants that fully capture the intention of the class, are much more time-consuming and consequently unfeasible to develop in most contexts [37]. In a multithreaded setting, however, invariants involving certain fields can only be upheld if these fields are only modified inside code sections that are executed atomically. In Java, the most common way to achieve atomicity is through methods that have the modifier **synchronized**, and thus use *monitors* associated with objects at runtime to ensure execution of code one thread at a time, with classical locks and semaphores available through libraries [164].

Even when a program's invariants and the associated atomicity requirements are understood separately from the actual code, there still remains the problem of splitting up the runtime state into units of concurrency. To distinguish between the sequential notion of object and active object, the unit of concurrency is most easily referred to as an actor, and the conversion process as *actorization*. Between one extreme, where all objects are included in one actor (thus losing all concurrency), and another extreme, where each object (possibly decomposed) becomes an actor, with significant additional messaging for synchronization needed as a result, there is a wide spectrum. For example, objects that encapsulate data structures such as lists, whose references are never visible outside the scope of the object they are used in, are arguably not candidates to become actors in their own right. A set of actors resulting from an actorization are similar to a set of concurrent object groups (cogs) in the CoBox model, where objects inside a cog collaboratively pass control around [183].

The research described in Paper V, summarized below, makes specific contributions to the first stage of actorization: determining whether invariants exist between fields in concurrent programs and specifying atomicity requirements separately from the code. This is done by analyzing runtime traces of legacy programs, i.e., by dynamic analysis of behavior. In Section 1.8.3, we describe current and future work to automate the full actorization process.

1.7 This Thesis: Contributions

1.7.1 Paper I: Location Independent Routing in Process Network Overlays

Any implementation of mobile active objects (or, actors) in a networked setting must consider the problem of location transparency. If active objects are statically allocated to nodes, communication between objects is easily reduced to communication between nodes, using, e.g., TCP/IP. An object identifier could then simply be represented by the IP address of the node at which it is located, along with some identifier unique in the scope of that node, such as a serial number. This is in effect what is done in the implementation of JCoBox with remoting, which uses Java's Remote Method Invocation mechanism [47]. The situation is quite different when objects are mobile: sending an inter-object message to some location which is only a temporary host of an object does not guarantee delivery, as is made clear in Section 1.3.

The network model used, as described in Section 1.5, is that of a connected graph of nodes with edges in the form of OSI layer 2 interconnects. Although the lower layers of the OSI model may have been formulated as a guide for physical implementors, they can also be realized, or simulated, using higher layers as abstractions that enable convenient organization of software. Hence, our network model is perhaps most easily implemented on top of TCP/IP, which we also did in our simulator described in Paper III, summarized in Section 1.7.3 below.

This paper uses what is possibly the smallest nontrivial asynchronous fragment of ABS, called μABS (micro-ABS) to demonstrate and analyze the approach of implementing active objects with location independent routing in OSI layer 2 networks. The absence of return values for procedures, i.e., the omission of a "return" statement for methods in classes of programs, means that two-sided communication sessions between objects is only possible if the sender (caller) passes its own identifier in a message. Then, the callee can invoke some method on the caller when done, in effect a callback routine, as is common in Erlang [155]. It can thus be argued that omission of return values is not a fundamental restriction.

To the best of our knowledge, this paper for the first time brings together ideas from content-centric networking, process algebras, and active objects to show the formal correctness of network-aware object behavior, in the form of contextual equivalence between a network-oblivious and a network-aware μABS semantics. To make the theoretical treatment more straightforward, no fundamental distinction is made between behavior related to program execution and behavior related to node management. Hence, processing of object and routing messages is done at the same level of abstraction as the processing of call messages. Arguably, this makes the paper somewhat less relevant to implementors, who may want to support many different languages in the same management infrastructure. Compared to Paper II, there are fewer language details to get in the way of the correctness analysis; the contextual equivalence argument is more distilled.

Statement of Contribution This paper was coauthored with Mads Dam. Ideas were discussed jointly, with both authors contributing to the writing of the paper. Mads wrote an initial draft, and Karl then assumed responsibility for detailed proofs, refined the text, and made many additions.

An earlier, shorter version of the paper was published in the proceedings of the 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP '14) [52]. The paper here is essentially the same as the paper version accepted for publication in the journal Service Oriented Computing and Applications [53]; only slight typesetting-related details differ.

1.7.2 Paper II: Efficient and Fully Abstract Routing of Futures in Object Network Overlays

One of the main differences between μABS and Core ABS is the absence of return statements of methods in the former, which necessitates using Erlang-style callbacks for mutual exchange of messages between objects. This paper extends μABS with futures as return values, bringing the resulting language, mABS (milli-ABS), much closer to Core ABS than μABS. As in Paper I, both a network-oblivious and a network-aware semantics is then provided, along with a correctness analysis based on contextual equivalence along the same lines.

In the network-oblivious reduction semantics of mABS (and Core ABS), a future is represented in a runtime configuration as a container having a unique identifier, f, and either a value or $\perp$. The former indicates that, at the configuration level, the future f is regarded as resolved, the latter that it is unresolved, as explained in Section 1.4.2. When a method call is dispatched, a new future container with $\perp$ is created. The receiver of the method call spawns a task, which produces a value which is put in the container. All computational tasks that have access to the future identifier can then retrieve, in an atomic action, the value from the centralized store in the container.

In a distributed setting based on asynchronous message passing, future values cannot be obtained directly in this way—the task that produces the value of a future may be at a different site than the task that will eventually retrieve the future. The problem of resolving and retrieving futures in programming language runtimes via message passing has been studied in the literature under the name *future update strategies* [90]. Three main strategies have been identified, which are characterized as either *eager* or *lazy* depending on whether update messages, containing values associated with futures, are delivered to all objects that have access to a future identifier, but do not necessarily attempt to retrieve the value, or *only* to those objects that attempt to retrieve the value. The first is an eager forward-based strategy, where objects that share future identifiers with other objects assume the obligation to share the resulting value, when possible. The second is an eager message-based strategy, where all recipients ("consumers") of future identifiers immediately register their interest with the object ("producer") that hosts the task computing the result of the associated method call; the producer object then sends out the value when available. Finally, in a lazy message-based strategy, objects request the future value they need at the point when it has been determined that the value is needed.

The paper applies the eager forward-based strategy to update futures in the mABS network-aware semantics. The primary reason is that the associated messaging load is then better dispersed among objects at runtime than for other strategies. When objects are allocated evenly between network nodes, as when using an object migration strategy from the paper summarized in Section 1.7.3, the messaging load related to futures becomes dispersed among nodes as well, which is desirable both from the point of view of maintaining decentralization and performance. Generally, an eager strategy is desirable to minimize latency if communicating objects can be many hops apart in the network. As a side-effect of depending on location independent routing, eager future message delivery also benefits from the low stretch of paths that results when routing tables are close to stable state.

A related Scala-based ABS backend [178, 148], supporting distribution without mobility, represents future containers as actors in their own right. One argument against this design is that future containers behave largely like memory cells, and are thus not *active* in the same sense as objects are. Yet, the distinction between active and passive in concurrency is not always clear [142]. However, in a mobile setting with performance objectives to uphold, having only one unit of mobility—the object—simplifies the dynamics.

The main differences between mABS and Core ABS are (1) the omission in mABS of an explicit syntax of functional terms and expressions, and an expression evaluation semantics, (2) the lack of a type system in mABS and accompanying safety theorem (subject reduction). With an implicit term and expression syntax, object identifiers and futures cannot be used as if they were terms in expressions, i.e., mixed in with other expression syntax. Consequently, while Core ABS values (ground terms) can contain an arbitrary (but bounded) number of futures, an mABS value is either a future, an object identifier, or an implicit ground term that contains neither object identifiers or futures. These omissions are motivated by the focus on the correctness argument, which is not fundamentally different, but more extensive, when expressions and types are explicitly included.

Statement of Contribution This paper was coauthored with Mads Dam. Both authors contributed to the writing of the paper, with Karl taking main responsibility for the formalization and proof details.

An earlier, shorter version of this paper was published in the proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! '13) [51]. Modulo typesetting-related details, this paper is the same as a longer paper submitted for journal publication.

1.7.3 Paper III: ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects

This paper provides a network-aware semantics for the full Core ABS language and explores how nondeterminism in executions can be restricted to perform runtime adaptation in terms of object mobility. The contextual equivalence proof with respect to the standard Core ABS semantics is not carried over from mABS, but little of fundamental interest is different in Core ABS. The combination of the Core ABS syntax, type system, expression evaluation semantics, and the network-aware semantics is referred to as ABS-NET.

The ABS-NET semantics is formulated to emphasize the separation between object behavior at runtime and node behavior. Conceptually, each node in the network has an interpreter layer, where objects execute (if any). The interpreter layer interacts with a node controller, where logic for message passing and load balancing resides. The interaction goes both ways. Migrating objects are received by the controller and placed in the interpreter layer, and outgoing object-addressed messages are dispatched to the controller to be routed to their destinations. The separation in the reduction semantics between nodes and objects makes clear what information is exchanged and when, going some way to suggest an Application Programming Interface (API) between implementations of a node controller and an interpreter layer.

The node controller running at each node decides, without involvement from centralized facilities, the scheduling of local objects and when to migrate objects to other nodes. In other words, the only knobs available for a controller are for object task execution and migration. Due to the asynchronous point-to-point message passing links, new objects and messages can arrive at any time.

We consider three performance objectives, in order of priority, for node controllers to achieve in cooperation: (1) node load, (2) link load, and (3) message latency. A node's load in a runtime configuration is defined as the number of objects located on the node which have an active task. The load of a link is the number of messages traversing it per unit of time. The latency of a particular message is the number of hops it traverses until it reaches its destination. To investigate how well these objectives can be achieved in a decentralized network, we use a TCP-based simulator for the ABS-NET semantics implemented in Java, parameterized on the migration strategy. The simulator introduces a degree of synchrony by executing node controller actions at an even pace.

The core migration strategy is based on coin-flipping for determining whether to migrate objects, and is based on a probabilistic algorithm with attractive convergence properties in fully connected graphs with synchronous rounds [17]. The idea is that, at intervals, node controllers attempt to migrate away some of their objects with active tasks by repeatedly picking a random neighbor whose load is known, and flipping a coin with a bias based on the node's own load and the neighbor's load to determine whether migration takes place. This avoids the oscillation that tends to happen with deterministic migration. To achieve progress towards minimizing link load, node controllers record the *affinity*, or communication intensity, of each object to other objects. When a random neighbor is selected and the coin flip determines that migration is to take place, the object with the highest affinity towards objects located in the direction of the neighbor is chosen. To account for the dynamics of communication patterns, the affinity added by observing a single message exchange decreases over time.

The results for grid, hypercube, and complete network topologies suggest that, at least for systems with long-running active objects, load can be balanced using the coin-flipping approach (even with a constant number of links per node), while making some headway towards meeting other performance objectives. One problem in sparser networks such as grids is that global minimization of some property may require intermediate allocations which are locally suboptimal.

Statement of Contribution This paper was coauthored with Mads Dam, Andreas Lundblad, and Ali Jafari. Karl developed and investigated the feasibility of the ABS-NET semantics of Core ABS, with input mostly from Mads. The quality of service criteria and outlines of the migration strategies were determined jointly. Karl implemented most of the strategies and evaluated them. Karl also wrote most of the paper.

This paper was, modulo minor typesetting changes, published in the proceedings of the 6th Interaction and Concurrency Experience (ICE '13) [159].

1.7.4 Paper IV: Decentralized Adaptive Power Control for Process Networks

In previous papers, the network of processing nodes, on which an ABS program executes, is assumed to be static throughout. Most immediately, static networks rule out node crashes, which would require use of replication to preserve network-oblivious behavior. However, the assumption also rules out benign dynamicity, in the form of controlled addition and shutdown of nodes. Even in a tentative cloud where nodes do not crash, there is still the need for nodes to be added or removed, or turned on and off, based on requirements on, e.g., computational load and energy consumption.

This paper introduces a largely language-independent model for programs executing in a benignly dynamic network, where nodes host mobile objects. The main contribution is a protocol for decentralized asynchronous networks that preserves objects and object-related messages located at nodes in the process of shutting down, by diverting them to other nodes. The protocol is reminiscent of a two-phase commit, in that nodes that decide to shut down send preparation messages to all their neighbors and require confirmation before proceeding to the next step. If a neighboring node simultaneously sends such preparation messages, a “winner” is eventually decided, and the loser reverts to normal state, able to make another attempt later. An abtract, bounded version of the protocol has been verified in an on-the-fly model checker, but this does not cover correctness in networks of unbounded size, which the network model permits. Consequently, we provide an alternative transition system model, defined in an inductive framework, to enable reasoning by induction about the safety side of correctness. A number of necessary properties are proven in this way, but fall short of establishing full correctness, which also includes liveness. This is left as future work.

As the protocol is formulated, with maximum nondeterminism, maintaining the connectedness of the network is left to the discretion of a scheduler. Although it does not in itself threaten the integrity of objects and object-related messages, a disconnected network must be avoided for program execution to proceed. Whether a network graph becomes disconnected or not through the removal of a single node can sometimes be determined locally, by knowledge of edges in the immediate neighborhood, but in the general case, it is a *global* property of the graph. Hence, when local, sound heuristics become inapplicable, a distributed algorithm which collects information from all nodes in the network must be used.

Finally, the paper discusses strategies for turning nodes on and off to achieve performance objectives, but a full evaluation in practice is left as future work.

Statement of Contribution This paper was coauthored with Mads Dam. Karl developed an outline of the shutdown protocol, modelled it in Spin, and refined it during model checking. Karl developed the transition system and the inductive proof approach. Karl wrote the paper. Mads contributed mainly through discussions and criticism.

The paper is previously unpublished, and has not been submitted for publication elsewhere.

1.7.5 Paper V: Dynamic Probabilistic Inference of Atomic Sets

Data-centric synchronization [62] is an approach to concurrency control in object-oriented languages where atomic access is specified directly for data in class fields and method arguments, as annotations. Synchronization is defined in terms of *atomic sets* of class fields, and methods that are *units of work* for atomic sets. The intention is that fields in the same atomic set have some, stated or unstated, invariant connecting them. A unit of work is a method which, when executed, guarantees atomic access to the instances of fields in the associated atomic sets. Finally, aliases allow atomic sets to cross object boundaries. In effect, these data-centric annotations comprise a statement of a program's concurrency semantics, with atomic sets capturing invariants and units of work capturing atomicity requirements.

In a shared-memory setting, how to best use concurrency primitives to implement this behavior can then be determined by a compiler rather than a programmer. A simple class definition in the AJ dialect of Java, annotated with atomic set annotations, is shown in Listing 1.3. For this particular class seen in isolation, the effect of adding the atomic set is equivalent to adding a **`synchronized`** modifier on the method `setNames()`.

```
public class Person {
  atomicset P;
  atomic(P) private String firstName;
  atomic(P) private String lastName;

  public void setNames(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

Listing 1.3: Java class annotated with atomic sets

As for formal verification of object-oriented programs, the major threat to eventual adoption of data-centric synchronization is the requirement to annotate existing code to facilitate use, in this case for Java programs. This paper addresses the problem of dynamic inference of atomic set-based annotations. The algorithm described in the paper, called Bait, uses Bayesian inference [163] to compute atomic sets, aliases, and units of work on-the-fly while a program executes. The algorithm records accesses to variables, and, for pairs of variables accessed subsequently, determines whether accesses are atomic. When this is the case, it is considered Bayesian *evidence* that the variables are connected by some (unknown) invariant. Over the course of the execution, evidence is weighed based on observed accesses; a data race counts against the existence of an invariant, but is not conclusive. One advantage of this approach is that it is not dependent on the specific mechanism used for concurrency control.

The Bait implementation is a tool chain that infers annotations in the AJ dialect for Java programs [62]. The tool chain consists of a byte code instrumenter and an inferencer. Given the byte code of a program, the instrumenter embeds code to record variable accesses and track necessary metadata during an execution. After the modified byte code is run, evidence of atomic sets and aliases are accumulated in data structures called belief configurations, which is finally used by the inferencer tool to derive annotations.

In the evaluation of the implementation, the annotations produced by Bait are compared to manual annotations for almost all AJ-annotated programs that are publicly available. The annotations are largely the same, with notable differences due to either bugs in the original annotations, or, in a small minority of cases, omissions by the tool. Two case studies are performed for the widely used Java open source programs Lucene [131] and Xalan [205], using benchmarks from the DaCapo benchmark suite [22], with favorable results.

Compared to other tools inferring atomicity specifications, Bait has an edge in that it (1) is robust against rare data races in traces, (2) takes distance between variable accesses into account, in terms of basic instructions, and (3) infers cross-object synchronization constraints in the form of aliases and **`unitfor`** annotations.

Data-centric synchronization allows for more fine-grained atomicity than Java's monitors, but coarser atomicity than locks or semaphores. Consequently, a program using synchronization based on atomic sets can be more concurrency-constrained, and thus execute with lower performance on multi-core computers. Related work referenced in the paper [203] shows that the slowdown need not be significant, however. Determining atomic sets is thus of interest both in its own right and for enabling actorization, as described in Section 1.6.3.

Statement of Contribution This paper was coauthored with Peter Dinges and Gul Agha. Karl tuned the algorithm and its parameters, did the evaluation, and helped clarify the algorithm's foundation in Bayesian inference theory. Karl wrote significant parts of the paper.

The paper is previously unpublished, and has not been submitted for publication elsewhere.

1.7.6 Other Papers

Karl has coauthored one paper on applications of distributed tree-based aggregation to search in a network of information [160]. Specifically, in NetInf [55], this approach is useful for continuous searches based on queries referencing metadata properties. Karl has also coauthored one paper on secure distributed top-k aggregation [106]. The topics and contents of these papers were considered outside the scope of this thesis, and they are therefore not included.

1.8 Conclusions and Future Work

1.8.1 Correctness and Decentralized Management of Distributed Objects

Paper I, II, and III demonstrate that location independent routing is a promising way to realize location independence for active objects. For Cloud Providers, the approach is useful both for virtual machines in IaaS clouds, or, more directly, for programs running in PaaS clouds. A central argument for location independence is that it enables resource management, and Paper III shows that balancing load is feasible using only migration of objects. As long as a migration strategy falls within the bounds of the state space established by the network-aware semantics for contextually equivalent execution, the result will be correct as far as a Cloud User can tell. Papers I to IV describe how Cloud Providers can perform decentralized resource management in clouds by migrating objects around and by turning nodes on and off. However, several other aspects of resource management remain to be considered in this setting, such as adjustment of buffer sizes, caches, memory usage, and processor load.

By virtue of the local nature of the reduction rules, there are no rules that simultaneously mutate the internal state of two nodes or more. Hence, each reduction rule that mutates node state corresponds to a function that takes current state as input and produces the new state. Assuming all elements of node state, such as routing tables, can be represented in an equivalent way in the semantics and an implementation, these functions can then be used directly in an implementation. In addition, as shown in the ABS-NET semantics, rules that mutate object state can be separated from rules that pertain to node state and message passing through links. The implementation of a language-specific interpreter layer can thus be done separately from a node implementation, with an interface along the lines of that in ABS-NET, and one node can run several different interpreters.

To ensure full implementation correctness down to the machine instruction level, the most straightforward approach is to encode a network-oblivious and a network-aware semantics in the logic of a proof assistant, e.g., Coq [43] or Isabelle [97]. Then, a machine-checked contextual equivalence proof can be made, and the corresponding implementation code can be extracted [119]. However, the code would still rely on its execution environment and primitives for message passing on the network. One possibility is to adapt work on verified operating systems, along the lines of seL4 [109], running virtualized inside a verified hypervisor [50]. The complexity of verifying the TCP/IP protocol suite [20] is an argument for relying on a non-IP network in such an environment.

The network-aware semantics for the ABS fragments in the papers are formulated to enable proving properties rather than to achieve high runtime performance. Hence, many optimizations to the sets of rules are possible:

- Unroutable object-addressed messages can be stored in a separate data structure, rather than sent to a self-loop link; when new routing information becomes available, the data structure can be searched for messages that have become routable.
- When an object sends a message, and the destination object is located on the same node (in the same interpreter layer), the message can be delivered directly to the destination without passing through the node controller at all; this includes self-calls.
- Object out-queues can be removed, replaced either by some general node message queue or by having the node synchronize with the object for outgoing messages; this makes runtime objects more similar to actors.
- The simple distance vector routing scheme can be replaced by a more sophisticated one, as long as it has similar self-stabilization properties.

For an optimized network-aware semantics to guarantee correct execution, contextual equivalence must be proved with the original semantics. The result is a hierarchy of refinements, capturing concerns ever closer to the implementation environment.

Besides adaptability of objects at runtime, there are other notions of adaptability, e.g., code adaptability, that can be added to a network-aware framework.

Program updates at runtime are already possible in, Erlang, and approaches in languages close to ABS can be adopted [208]. ABS has been extended with *deployment components* [105] to allow programmers to control resource usage at runtime. These extensions can be implemented in an ABS-NET runtime, with nodes acting as deployment components. The result is in effect a reflective middleware layer [111].

1.8.2 Computation in Dynamic and Adversarial Networks

Scaling a processing network up and down in a controlled way is made possible through the shutdown protocol in Paper IV. Still, failures such as node crashes are not considered. An important issue is whether to consider node crashes and link failures separately. If only node crashes are considered, messages in links going to and from a crashing node are lost. The principal theory for dealing with failures in distributed systems, *failure detectors* [36], is most straightforwardly considered at the level of nodes. Generally, it cannot be assumed that node crashes and link failures are distinguishable, and thus, can be dealt with differently. Arguably, locally at a node, a permanently lost link to another node is indistinguishable from that node crashing. In addition, a falsely reported crash from an imperfect detector requires the same measures—e.g., recovery and retransmission of messages—as a temporary link failure. Hence, a network model with eventually consistent failure detectors seems appropriate for capturing robustness against both node crashes and link failures in practice. It still remains, however, to investigate how and when replication is to be used to preserve objects and messages, to uphold the correspondence with a non-networked execution. The state checkpointing approach of Field and Varela for actors is one option [69].

Dealing with nondeterministic failures in this way still assumes that nodes do not deviate from established protocols when communicating with neighbors. In a model with Byzantine failures [116], such deviations are possible. One way to deal with Byzantine failures is by using cryptographic primitives to, e.g., certify integrity of object states and object-addressed messages. Designers of content delivery networks address similar problems [188], but for *passive* objects such as video and audio files, rather than active objects. Inlined monitors and proof-carrying code [132] can ensure programs received from potentially unreliable sources behave as expected. Finally, privacy may be a concern for both Cloud Users and End Users, and is left as future work.

1.8.3 Data-Centric Synchronization and Actorization

Paper V provides one piece of a solution to the problem of making programs written in object-oriented mainstream languages ready to execute in PaaS clouds using message-passing concurrency control. Suppose the concurrency semantics of some program is captured precisely and accurately with data-centric annotations. For reasonable performance, even with a lock-based implementation, it may still be necessary to perform various refactorings. For example, a class having multiple atomic sets (which by definition are disjoint) can be decomposed into several classes, each with a single atomic set of fields, with methods divided according to which fields are accessed. The annotations can then be used to produce an *actorized* version of the program, ready for execution in a PaaS cloud. In the actorized program, objects are still not necessarily mapped to actors one-to-one, because of the presence of aliases between atomic sets in different classes. Generally, the problem is one of correctly partitioning runtime objects into disjoint subsets, each with separate control flow and local data, so that when an object in one subset calls a method on an object in another subset, this is translated to asynchronous message passing, as in ABS. Two possible semantics-preserving partitions of a collection of objects can be different in terms of degree of concurrency, and requirements on communication and synchronization. Decomposing tightly connected cliques of objects into different sets inevitably leads to more messaging, but can speed up execution on multi-core machines.

In finding a robust theoretical description of the actorization process, Java itself is not an ideal language as a starting point. Java makes no fundamental distinction between mutable and immutable objects, and object references are not necessarily serializable or accessed through interfaces. One option is to define a subset of the Java-like Jinja language [110] with a formalized threading model [126], or define a Java-like threaded version of ABS with monitors and locks. To properly capture the semantics of **`unitfor`** annotations in methods, which can join together atomic sets in otherwise unrelated objects, additional synchronization must be used, e.g., in the form of two-phase commits between actors.

After determining the source language, a pipeline for actorization can be defined, as follows. After performing annotation inference, a program can be annotated with atomic sets, aliases, and units of work, and all control-centric concurrency primitives removed. Then, the program can either be executed in the threaded model through a compiler (similar to the now defunct AJ compiler [62]), or be converted to a language where boundaries between actors and objects are made clear. JCoBox is one example of such as language, in that when an object is created, it is either part of the same concurrent object group (cog) that spawned it, or becomes a new cog. If an ABS-like language is used, the language can be directly executable in an implementation along the lines of ABS-NET. Alternatively, the program can be converted into another language and have actor-spawning primitives replaced by API calls to an actor library, such as Akka for Java [6]. A conceptual overview of the process is shown in Figure 1.16.

To illustrate the process in greater detail, consider the Java program in Listing 1.4, derived from Paper V, which uses monitors to prevent concurrent access to a list of Uniform Resource Locators (URL) in objects of the class `Downloadmanager`. In the `main()` method, located in the class `Download`, a number of URLs are added to the list, and then retrieved by either of two worker threads of class `DownloadThread`, running in parallel, for download.

Ideally, after annotation inference, and removal of monitors by dropping the **`synchronized`** modifier for the `getNextURL()` method in the class `DownloadManager`,

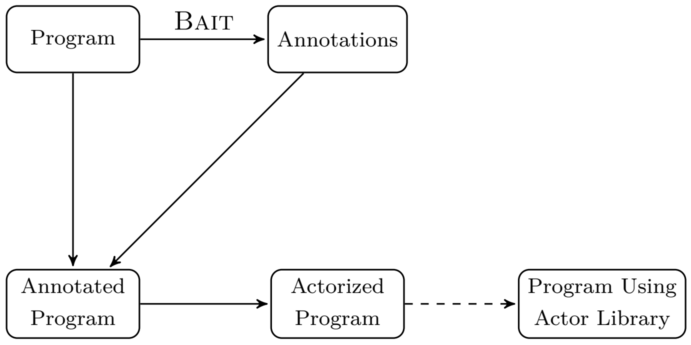


Figure 1.16: Proposed actorization process

the AJ program in Listing 1.5 is produced. Note that there is an alias from the atomic set `U` in `DownloadManager` to the atomic set `L` in class `List`, which ensures that the method `getNextURL()` atomically accesses the associated `List` instance.

Since there is no obvious need for decomposition of classes, the program in Listing 1.5 can be used to produce the actorized JCoBox-like program in Listing 1.6, where some object instantiations in the `main()` method now include the keyword **`actor`**. In fact, the classes in these instantiations either have an atomic set in Listing 1.5, or are subclasses of `Thread`. Because of the alias between the classes, the `List` instance in `DownloadManager` is not initialized in this way. Note that the actorized classes, `DownloadManager` and `DownloadThread`, are now accessed through interfaces, extracted in the obvious way.

To enable running the program in Listing 1.6, one possibility is to do a source-to-source translation, e.g., using abstract syntax tree transformation in a framework such as JastAddJ [64], to a program that uses the Akka actor library. Assume a statically imported method `create()`, that takes as arguments an interface and an Akka `Creator` object that sets up an actor-wrapped object instance. The actorized `Download` class then becomes as shown in Listing 1.7; the remaining classes are unchanged from Listing 1.6.

In summary, the CoBox model of concurrent object groups, while not ideal to implement at the network level, appears to be a useful as an intermediate level of organization between sequential objects and actors.

Data-centric annotations are useful also for local execution, and one important task is therefore the development of an efficient lock-based AJ compiler to replace the defunct Eclipse-based prototype compiler [62]. Such a compiler is useful as a point of reference when running actorized programs, and for extending the use of data-centric annotations beyond Java. As actor libraries become mature, the performance differences between a lock-based version of a program annotated with atomic sets, and the corresponding actorized version, can be expected to decrease, at least for programs with coarse-grained synchronization.

```java
class List {
  int size;
  Object[] elements;
  /* ... */
  public int size() { return this.size; }
  public Object get(int index) {
    if (0 <= index && index < this.size) {
      return this.elements[index];
    } else { return null; }
  }
  /* ... */
}
class DownloadManager {
  // Atomic access ensured by monitors
  List urls;
  /* ... */
  public synchronized URL getNextURL() {
    if (this.urls.size() == 0) return null;
    URL url = (URL) this.urls.get(0);
    this.urls.remove(0);
    announceStartInGUI(url);
    return url;
  }
  /* ... */
}
class DownloadThread extends Thread {
  DownloadManager manager;
  /* ... */
  public void run() {
    URL url;
    while((url = this.manager.getNextURL()) != null) {
      download(url); // Blocks while waiting for data
    }
  }
  /* ... */
}
public class Download {
  public static void main(String[] args) {
    DownloadManager manager = new DownloadManager();
    for (int i = 0; i < 31; i++) {
      manager.addURL(new URL("http://www.example.com/file" + i));
    }
    DownloadThread thread1 = new DownloadThread(manager);
    DownloadThread thread2 = new DownloadThread(manager);
    thread1.start();
    thread2.start();
  }
}
```

Listing 1.4: Java program using monitors

```
class List {
  atomicset L;
  atomic(L) int size;
  atomic(L) Object[] elements;
  /* ... */
  public int size() { return this.size; }
  public Object get(int index) {
    if (0 <= index && index < this.size) {
      return this.elements[index];
    } else { return null; }
  }
  /* ... */
}
class DownloadManager {
  atomicset U;
  atomic(U) List urls|L=this.U|;
  /* ... */
  public URL getNextURL() {
    if (this.urls.size() == 0) return null;
    URL url = (URL) this.urls.get(0);
    this.urls.remove(0);
    announceStartInGUI(url);
    return url;
  }
  /* ... */
}
public class DownloadThread extends Thread {
  DownloadManager manager;
  /* ... */
  public void run() {
    URL url;
    while((url = this.manager.getNextURL()) != null) {
      download(url); // Blocks while waiting for data
    }
  }
  /* ... */
}
public class Download {
  public static void main(String[] args) {
    DownloadManager manager = new DownloadManager();
    for (int i = 0; i < 31; i++) {
      manager.addURL(new URL("http://www.example.com/file" + i));
    }
    DownloadThread thread1 = new DownloadThread(manager);
    DownloadThread thread2 = new DownloadThread(manager);
    thread1.start();
    thread2.start();
  }
}
```

Listing 1.5: AJ program without monitors

```
class List {
 int size;
 Object[] elements;
 /* ... */
 public int size() { return this.size; }
 public Object get(int index) {
   if (0 <= index && index < this.size) {
     return this.elements[index];
   } else { return null; }
 }
 /* ... */
}
class DownloadManager implements IDownloadManager {
 List urls;
 /* ... */
 public URL getNextURL() {
   if (this.urls.size() == 0) return null;
   URL url = (URL) this.urls.get(0);
   this.urls.remove(0);
   announceStartInGUI(url);
   return url;
 }
 /* ... */
}
public class DownloadThread extends Thread implements IDownloadThread {
 IDownloadManager manager;
 /* ... */
 public void run() {
   URL url;
   while((url = this.manager.getNextURL()) != null) {
     download(url); // Blocks while waiting for data
   }
 }
 /* ... */
}
public class Download {
 public static void main(String[] args) {
   // Actorized initializations
   IDownloadManager manager = new actor DownloadManager();
   for (int i = 0; i < 31; i++) {
     manager.addURL(new URL("http://www.example.com/file" + i));
   }
   IDownloadThread thread1 = new actor DownloadThread(manager);
   IDownloadThread thread2 = new actor DownloadThread(manager);
   thread1.start();
   thread2.start();
 }
}
```

Listing 1.6: JCoBox-like actorized program

```
public class Download {
  public static void main(String[] args) {
    final IDownloadManager manager =
      create(IDownloadManager.class,
        new Creator<IDownloadManager>() {
          @Override
          public IDownloadManager create() throws Exception {
            return new DownloadManager();
          }
        });
    for (int i = 0; i < 31; i++) {
      manager.addURL(new URL("http://www.example.com/file" + i));
    }
    IDownloadThread thread1 =
      create(IDownloadThread.class,
        new Creator<IDownloadThread>() {
          @Override
          public IDownloadThread create() throws Exception {
            return new DownloadThread(manager);
          }
        });
    IDownloadThread thread2 =
      create(IDownloadThread.class,
        new Creator<IDownloadThread>() {
          @Override
          public IDownloadThread create() throws Exception {
            return new DownloadThread(manager);
          }
        });
    thread1.start();
    thread2.start();
  }
}
```

Listing 1.7: `Download` class with Akka-based actor initialization

1.8.4 Tradeoffs Between Asynchrony and Accuracy

The successful application of round-based, synchronous algorithms for decentralized, asynchronous load balancing (without synchronizers [13]) opens up the question of the tradeoffs between, on one hand, timeliness and accuracy, and on the other hand, the amount of synchrony. Besides for load balancing, similar concerns apply for stabilization of routing tables, and for distributed aggregation. A conjecture is that, at least for specific problems, there are precise requirements of synchrony to achieve specific bounds on convergence, or of result accuracy.

Chapter 2

Location Independent Routing in Process Network Overlays

Mads Dam   Karl Palmskog

KTH Royal Institute of Technology, Sweden
{mfd, palmskog}@kth.se

Abstract

In distributed computing, location transparency—the decoupling of objects from their physical location—is desirable in that it can simplify application development and enable efficient resource allocation. Many systems for location transparency are built on TCP/IP. We argue that addressing mobile objects in terms of temporary hosts may not be the best design decision. Object migration makes it necessary to use dedicated routing infrastructures, e.g., location servers, to deliver inter-object messages. This incurs high costs in terms of complexity, overhead, and latency. Here, we defer object overlay routing to a networking layer, by replacing TCP/IP with a location independent routing scheme which directs messages to destinations determined by flat identifiers instead of IP addresses. Consequently, messages are delivered directly to objects, instead of possibly out-of-date locations. We explore the scheme using a small object-based language with asynchronous message passing, similar to Core Erlang. We provide a standard, network-oblivious operational semantics of this language, and a network-aware semantics which accounts for many aspects of distribution and routing. The main result is that program execution on top of an abstract network of processing nodes connected by asynchronous point-to-point communication channels preserves network-oblivious behavior in a sound and fully abstract way, in the sense of contextual equivalence. This is a novel and strong result for such a low-level model. Previous work has addressed distributed implementations only for fully connected TCP underlays, where contextual equivalence is typically too strong, due to the need for locking to resolve preemption arising from mobility.

2.1 Introduction

The decoupling of computational tasks, objects, or virtual machines from their physical realization, and particularly their location, can be beneficial for both application developers and providers of computing infrastructures. If tasks are adaptively allocated to nodes in a processing network, application developers can avoid explicit resource management, and infrastructure providers can achieve high utilization while fulfilling requirements such as power consumption and response time. This approach can result in simpler application logic, better service quality, and, ultimately, lower cost of development, operations, and management. The question is how to realize this potential with minimal overhead, and in such a way that applications behave predictably.

A key problem is how to handle object and task mobility in an efficient manner. Since the allocation of objects to nodes needs to be dynamic, some form of message routing is needed to ensure that inter-object messages reach their destinations quickly, and with minimal overhead. Various approaches have been considered in the literature; Sewell et al. provide a comprehensive survey [184]. One option is to maintain a centralized or distributed database of object locations. Such a database can be used for both forwarding, by routing messages through the forwarding server, and for location querying, by using the database to look up destination object locations. In either case, object location and the location database must be kept consistent, which requires synchronization. Many experimental object mobility systems in the literature use some form of replicated or distributed location databases [63, 19, 87, 184]. Another option is for nodes to maintain forwarding pointers, as in the Emerald system [107]. Migration then causes forwarding pointer chains to be extended by one further hop, and some mechanism is typically used to piggyback location update information onto messages, to ameliorate forwarding chain growth. This mechanism is used, for instance, in the JoCaml programming language [42]. Many solutions involve some form of broadcast or multicast search. For instance, an object may use multicasting to find an object if a pointer for some reason has become stale, as in Emerald, or for service discovery, as in Jini [11]. Other solutions have been explored too, such as tree-structured DNS-like location directories [198], Awerbuch and Peleg's distributed directories [14], and Demmer and Herlihy's arrow protocol [57].

We argue that the main source of the difficulties these approaches are designed to solve is the distinction between destination host identifier (e.g., IP address) and search identifier (e.g., object identity). In a fully mobile setting, the location at which an object resides has no intrinsic interest[1]. What is of interest is the message destination, i.e., that an RPC destined for the object with identity o is routed to the location where o resides, and not somewhere else. In other words, we suggest that inter-object message routing should really be done using the destination object identity. An assumed host can for all the sender knows be out-of-date.

[1]Location has interest as a source of latency, for instance, but that is another matter.

In the networking community, variations of this idea have been the subject of significant attention over the last decade. There are several proposals to replace the location-based routing of traditional IP networks with *location independent* schemes that route messages according to names, or content. Names can be flat, unstructured identifiers [27], or they can encode some form of signed content identity, as in content-centric networking [100]. The general goal is to devise routing schemes for flat name spaces that are *compact*, i.e., such that routing tables can be represented at each node using space sublinear in the number of destinations, and such that path lengths, and hence message latencies, do not grow too far from the optimal. This requirement of low *stretch*, defined as the ratio of route length to shortest path length, precludes the use of location registers and hierarchical IP-like naming schemes.

In this paper, we examine location independent routing in the context of a rudimentary language of message-passing objects, in the style of Core Erlang [29], and propose a formal, maximally nondeterministic, network-aware runtime semantics of this language. The main purpose is to show that this routing scheme offers a new space for solutions to the object mobility problem with some attractive properties:

Minimalism A whole swathe of software becomes superfluous, which manages address lookups, message forwarding, rerouting, and address bookkeeping, and the synchronization overhead between location registers and the migrating objects is eliminated. As a result, the "trusted computing base" of the networked execution platform is significantly reduced, both in terms of size and complexity.

Decentralization No centralized or decentralized object location database is required, since the routing mechanism itself ensures that inter-object messages are routed to the proper host.

Low stretch Traffic overhead is decreased. First, mobility support on top of IP needs to perform routing both at the IP and at the application level. Name-based routing in effect eliminates the need for IP-level routing. Moreover, in steady state, the simple distance vector (d.v.) routing scheme we use has stretch 1, so message delivery overhead is minimal. (However, d.v. routing is not compact: routing tables have size $\Omega(n)$, with n the number of destinations. We leave such scalability issues for future work.)

Self-stabilization In faulty situations, if connection to a location register is lost, message delivery becomes impossible (or possible only through costly mechanisms such as broadcasting, as in Emerald or Jini). Routing can be made self-stabilizing, and thus be able to adapt to any type of disturbance, as long as connectivity is maintained. This allows computation to progress, including delivery of messages and migration of objects, even when the network is under considerable churn.

The scheme we propose assumes only a network graph with OSI layer 2 connectivity, i.e., the possibility of non-lossy, ordered communication between neighboring nodes. Such a graph can be realized in many ways—for instance, as a physical non-IP network, or as a TCP-based virtual network overlay with some desired topology. A hybrid approach, were our scheme runs without TCP/IP for physically connected network nodes inside datacenters, and use TCP-based bridging to transport messages between different datacenters, is one way to carry out an implementation in current networks.

The object-based language defined in Section 2.3, μABS, is a rudimentary fragment of the ABS (Abstract Behavioral Specification) language [103] used in the EU FP7 project HATS for studying phenomena related to software evolvability and adaptation. μABS includes a minimal set of features for method invocation, object creation, and sequential control, which is sufficient, however, to allow simple object-oriented programs to be programmed in a natural way. In Section 2.4, we give an initial "reference" semantics of μABS in the style of rewriting logic [40], which does not take into account aspects related to location, naming, routing, or communication. The second semantics in Section 2.5 takes these aspects into account by describing program execution on top of an arbitrary, but concrete, processor network where nodes are connected point-to-point by asynchronous channels. With the help of runtime configuration normal forms, defined in Section 2.6, we state the main result in Section 2.7: that the network-aware semantics is sound and fully abstract with respect to the reference semantics. We base the analysis on contextual equivalence [169], which requires a witness relation that preserves some primitive observations, here calls to the outside world, in both directions, and is preserved under weak reductions and when applying a context.

In an implementation of the network-aware semantics, many choices left open through the use of unrestricted nondeterminism must be resolved. Hence, in Section 2.8, we discuss scheduling. Finally, Section 2.9 describes related work, and Section 2.11 concludes. All proofs are deferred to Section 2.12.

2.2 Notation

We use a vector notation to abbreviate sequences, for compactness. Thus, $\overline{x}$ abbreviates $x_1, \ldots, x_n$, possibly empty; $x_0, \overline{x}$ abbreviates $x_0, \ldots, x_n$. Let $g : A \to B$ be a finite map. The update operation for g is defined by $g[b/a](x) = g(x)$ if $x \neq a$ and $g[b/a](a) = b$. We use $\perp$ for bottom elements, and $A_\perp$ for the lifted set with partial order $\sqsubseteq$ such that $a \sqsubseteq b$ if and only if either $a = b \in A$ or else $a = \perp$. Also, if x is a variable ranging over A, we often use $x_\perp$ as a variable ranging over $A_\perp$. For g a function $g : A \to B_\perp$ we write $g(a) \downarrow$ if $g(a) \in B$, and $g(a) \uparrow$ if $g(a) = \perp$. The product of sets (flat CPOs) A and B is $A \times B$ with pairing (a, b) and projections π_1 and π_2.

2.3 The μABS Language

We define μABS, short for micro-ABS, a small, distributed, object-based language with asynchronous method calls. Its syntax is shown in Table 2.1.

$x, y \in \mathit{Var}$			Variable
$e \in \mathit{Exp}$			Expression
$C, m \in \mathit{SID}$			Static identifier
P	$::=$	$\overline{CL}\ \{\overline{x}, s\}$	Program
CL	$::=$	$\textbf{class}\ C(\overline{x})\{\overline{y}, \overline{M}\}$	Class definition
M	$::=$	$m(\overline{x})\ \{\overline{y}, s\}$	Method definition
s	$::=$	$s_1; s_2 \ \mid x = \mathit{rhs} \mid \textbf{skip} \mid e!m(\overline{e})$ $\mid \textbf{if}\ e\ \{s_1\}\ \textbf{else}\ \{s_2\} \mid \textbf{while}\ e\ \{s\}$	Statement
rhs	$::=$	$e \mid \textbf{new}\ C(\overline{e})$	Right-hand side

Table 2.1: μABS abstract syntax

Programs are sequences of class definitions, appended with a sequence of variables $\overline{x}$, and a "main" statement s, which can use those variable to set up an initial collection of objects. The class hierarchy is flat and fixed. Objects have parameters $\overline{x}$, local variable declarations $\overline{y}$, and methods $\overline{M}$. Methods have parameters $\overline{x}$, local variable declarations $\overline{y}$ and a statement body s. For simplicity, we assume that variables have unique declarations. The definition of expressions e is left open, but we require that expression evaluation can be made side-effect free. We omit types from the presentation—types could be added, but would not affect the results of the paper in any significant way.

Besides the standard sequential control structures, statements include constructs for asynchronous method invocation and object creation. Sequential composition is associative with unit **skip**, i.e., the statements $s;\textbf{skip}$, $\textbf{skip};s$, and s are identified. Methods lack return statements; method bodies are simply evaluated to the end, at which point the evaluating task is removed. In the absence of return statements, objects can communicate using callbacks in a manner similar to interprocess communication in Erlang, as illustrated in Example 2.3.1. In other work, we have extended the language with lazy return values for method calls in the form of futures, making the analysis more involved [51].

Example 2.3.1. The μABS program in Listing 2.1 constructs an object ring with (here) 42 elements. After the method `serve` is invoked on a server object, values are circulated in the ring that is being constructed, finally resulting in the computation of `foo(...foo(foo(42,42),41)...,1)`. When it passes its value, each ring cell decrements the counter `i`, which is initialized to the value 42 when received by the first cell. The final cell returns the final value to the server, which sends it to the client. The client then finally calls the `output` method on the reserved object identifier `ext`.

```
class Server() { ,
  serve(from, x) { c,
    c = new Cell(self, from);
    c!process(x, x)
  }
  return(to, result) {
    to!response(result)
  }
}

class Client(arg) { ,
  use(server) { ,
    server!serve(self, arg)
  }
  response(y) { ,
    ext!output(y)
  }
}
{ server, client,
  server = new Server();
  client = new Client(42);
  client!use(server)
}
```

```
class Cell(root, to) { ,
  process(x, iter) { c, r,
    if iter = 0 {
      root!return(to, x)
    } else {
      c = new Cell(root, to);
      r = foo(x, iter);
      c!process(r, iter-1)
    }
  }
}
```

Listing 2.1: μABS ring program

2.4 Reference Semantics

We first present a standard reduction semantics for μABS in the style of rewriting logic [40], which is important as the point of reference for later refinements. The semantics uses a reduction relation $cn \to cn'$ where cn and cn' are *configurations*, as determined by the runtime syntax in Table 2.2. Later on, we introduce different configurations and transition relations, and so use index 1, or mention, e.g., configurations of "type 1" for this first semantics when we need to disambiguate.

With respect to the runtime syntax, $\preceq$ is the subterm relation, and we use disjoint, denumerable sets of object identifiers $o \in OID$ and primitive values $p \in PVal$. Lifted values are ranged over by $v_\perp \in Val_\perp$. We often refer to OIDs as *names*, and bind OIDs using the binder bind, which is reminiscent of the restriction binder in π-calculus [144]. The free names of the configuration cn is the set $\mathtt{fn}(cn)$, and $OID(cn)$ is the set of OIDs of object containers occurring in cn. Standard alpha congruence applies to name binding. Later on, in the type 2 semantics, we drop name binding entirely.

Configurations are multisets of containers of which there are three types: tasks, objects, and calls. Configuration juxtaposition is assumed to be commutative and associative with unit 0. In addition, we assume the standard structural identities $\mathsf{bind}\ o.0 = 0$ and $\mathsf{bind}\ o.(cn_1\ cn_2) = (\mathsf{bind}\ o.cn_1)\ cn_2$ whenever $o \notin \mathtt{fn}(cn_2)$. We often use a vectorized notation $\mathsf{bind}\ \overline{o}.cn$ as abbreviation, letting $\mathsf{bind}\ \varepsilon.cn = cn$,

$x \in Var$			Variable
$o \in OID$			Object identifier
$p \in PVal$			Primitive value
$v \in Val$	$=$	$PVal \cup OID$	Value
$l \in TEnv$	$=$	$Var \to Val_\perp$	Task environment
$a \in OEnv$	$=$	$Var \cup \{\mathbf{self}\} \to Val_\perp$	Object environment
$tsk \in Tsk$	$::=$	$\mathsf{t}(o, l, s)$	Task
$obj \in Obj$	$::=$	$\mathsf{o}(o, a)$	Object
$call \in Call$	$::=$	$\mathsf{c}(o, m, \overline{v})$	Call
$ct \in Ct$	$::=$	$tsk \mid obj \mid call$	Container
$cn \in Cn$	$::=$	$0 \mid ct \mid cn\ cn' \mid \mathsf{bind}\ o.cn$	Configuration

Table 2.2: μABS type 1 runtime syntax

where ε is the empty sequence. The structural identities then allow us to rewrite each configuration into a *standard form* $\mathsf{bind}\ \overline{o}.cn$, such that each member of $\overline{o}$ occurs free in cn, and cn has no occurrences of the binding operator bind.

Tasks are used for method body elaboration. Task and object environments l and a, respectively, map local variables to assignable values. Object environments are aware of a special variable **self**, which is mapped to the identifier of the associated object. Upon method invocation, a task environment is initialized using the operation $\mathtt{locals}(o, m, \overline{v})$ by mapping the formal parameters of the method m in the class of o to the corresponding actual parameters in $\overline{v}$, and initializing the task-local variables to suitable null values. Object environments are initialized with the operation $\mathtt{init}(C, \overline{v}, o)$, which maps the parameters of C to $\overline{v}$, initializes the object-local variables as above, and maps **self** to o. In addition to `locals` and `init`, we use the auxiliary operation $\mathtt{body}(o, m)$, which retrieves the statement of the shape s in the definition body for method m in the class of o, and $[\![e]\!]_{(a,l)} \in Val$ is used for evaluating the expression e in the object environment a and task environment l to a value.

Figure 2.1 presents the reduction rules, using the notation $cn \vdash cn' \to cn''$ as shorthand for $cn\ cn' \to cn\ cn''$. The reduction rules obey some basic sanity properties.

Proposition 2.4.1. *Suppose $cn \to cn'$. Then, the following holds:*

1. $\mathtt{fn}(cn') \subseteq \mathtt{fn}(cn)$.
2. *If $\mathsf{o}(o, a) \preceq cn$, then $\mathsf{o}(o, a') \preceq cn'$ for some object environment a'.*

Executions of programs in the semantics are sequences of configurations derived by the rules, starting from an initial configuration, as defined below.

Definition 2.4.2 (Type 1 Initial Configuration)**.** Suppose we are given a program $\overline{CL}\ \{\overline{x}, s\}$, and that o_{main} is a reserved object identifier different from ext. Then, CTXT-1: If $cn_1 \to cn_2$, then $cn \vdash cn_1 \to cn_2$

CTXT-2: If $cn_1 \to cn_2$, then $\mathsf{bind}\ o.cn_1 \to \mathsf{bind}\ o.cn_2$

WLOCAL: If $x \in \mathtt{dom}(l)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o,a) \vdash \mathsf{t}(o,l,x = e; s) \to \mathsf{t}(o,l[v/x],s)$

WFIELD: If $x \in \mathtt{dom}(a)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o,a)\ \mathsf{t}(o,l,x = e; s) \to \mathsf{o}(o,a[v/x])\ \mathsf{t}(o,l,s)$

SKIP: $\mathsf{t}(o,l,\mathbf{skip}) \to 0$

IF-TRUE: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\mathsf{o}(o,a) \vdash \mathsf{t}(o,l,\mathbf{if}\ e\ \{s_1\}\ \mathbf{else}\ \{s_2\}; s) \to \mathsf{t}(o,l,s_1;s)$

IF-FALSE: If $[\![e]\!]_{(a,l)} = 0$, then
$\mathsf{o}(o,a) \vdash \mathsf{t}(o,l,\mathbf{if}\ e\ \{s_1\}\ \mathbf{else}\ \{s_2\}; s) \to \mathsf{t}(o,l,s_2;s)$

WHILE-TRUE: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\mathsf{o}(o,a) \vdash \mathsf{t}(o,l,\mathbf{while}\ e\ \{s_1\}; s) \to \mathsf{t}(o,l,s_1;\mathbf{while}\ e\ \{s_1\};s)$

WHILE-FALSE: If $[\![e]\!]_{(a,l)} = 0$, then
$\mathsf{o}(o,a) \vdash \mathsf{t}(o,l,\mathbf{while}\ e\ \{s_1\}; s) \to \mathsf{t}(o,l,s)$

CALL-SEND: Let $o' = [\![e_1]\!]_{(a,l)}$, $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$ in
$\mathsf{o}(o,a) \vdash \mathsf{t}(o,l,e_1!m(\overline{e_2}); s) \to \mathsf{t}(o,l,s)\ \mathsf{c}(o',m,\overline{v})$

CALL-RCV: Let $l = \mathtt{locals}(o,m,\overline{v})$, $s = \mathtt{body}(o,m)$ in
$\mathsf{o}(o,a) \vdash \mathsf{c}(o,m,\overline{v}) \to \mathsf{t}(o,l,s)$

NEW: Let $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \mathtt{init}(C,\overline{v},o')$ in $\mathsf{o}(o,a) \vdash$
$\mathsf{t}(o,l,x = \mathbf{new}\ C(\overline{e}); s) \to \mathsf{bind}\ o'.\mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a')$

Figure 2.1: μABS type 1 reduction rules

a *type 1 initial configuration* cn_{init} for the program has the shape

$$\mathsf{bind}\ o_{main}.\mathsf{o}(o_{main},\perp)\ \mathsf{t}(o_{main},l_{init},s)$$

where $\perp$ is the everywhere undefined initial object environment and l_{init} is the initial task environment assigning default values to the variables $\overline{x}$.

Definition 2.4.3 (Type 1 Well-formedness)**.** A configuration cn is *type 1 well-formed* (WF1) if cn satisfies:

1. *OID Uniqueness*: If $\mathsf{o}(o_1,a_1), \mathsf{o}(o_2,a_2) \preceq cn$ are distinct object occurrences in cn, then $o_1 \neq o_2$.

2. *Task-Object Existence*: If $\mathsf{t}(o,l,s) \preceq cn$, then $\mathsf{o}(o,a) \preceq cn$ for some object environment a.

We note that well-formedness holds for initial configurations and is preserved under reduction.

Proposition 2.4.4 (WF1 Preservation)**.** *Let cn be a type 1 configuration. Then, the following holds:*

1. *If cn is a type 1 initial configuration, then cn is WF1.*
2. *If cn is WF1 and $cn \to cn'$, then cn' is WF1.*

Well-formedness preservation is important, as it ensures that objects, if defined, are defined uniquely, and that there are no dangling tasks.

Our approach to implementation correctness is based on the notion of contextual equivalence [169]. The goal is to show that it is possible to remain strongly faithful to the reference semantics in a networked setting, while leaving nondeterminism to be handled by a separate scheduler. This allows us to draw strong conclusions also in the case when a scheduler is added, as described in Section 2.8. Contextual equivalence requires of a pair of equivalent configurations, first, that the internal transition relation $\to$ is preserved in both directions, and second, that the relation is preserved when applying a context configuration, while preserving a set of external observations. A number of works [101, 180] have established strong relations between contextual equivalence for reduction oriented semantics and bisimulation/-logical relation based equivalences for sequential and higher-order computational models.

Assume an OID *ext* representing the "outside world", not allowed to be bound or defined in well-formed configurations. An observation, or *barb*, is a method call to *ext* with evaluated arguments, of the shape $ext!m(\overline{v})$, and ranged over by *obs*. The observation predicate $cn \downarrow obs$ is defined to hold just in case $cn = \mathsf{bind}\ \overline{o}.cn'\ \mathsf{c}(ext, m, \overline{v})$ for some cn'. The derived predicate $cn \Downarrow obs$ holds just in case $cn \to^* cn' \downarrow obs$ for some cn'.

Definition 2.4.5 (Type 1 Witness Relation, Type 1 Contextual Equivalence)**.** Let $\mathcal{R}$ range over binary relations on type 1 well-formed configurations. The relation $\mathcal{R}$ is a *type 1 witness relation*, if $cn_1\ \mathcal{R}\ cn_2$ implies

1. *Reduction Closure*: If $cn_1 \to cn_1'$, then $cn_2 \to^* cn_2'$ for some cn_2' such that $cn_1'\ \mathcal{R}\ cn_2'$.
2. *Context Closure*: If $cn_1\ cn$ is WF1, then $cn_2\ cn$ is WF1 and $cn_1\ cn\ \mathcal{R}\ cn_2\ cn$.
3. *Barb Preservation*: If $cn_1 \downarrow obs$, then $cn_2 \Downarrow obs$.

Additionally, the converse properties must hold with $\mathcal{R}^{-1}$ for $\mathcal{R}$ above. We define *type 1 contextual equivalence*, $\simeq_1$, as the union of all type 1 witness relations. Additionally, we say that the WF1 configurations cn_1 and cn_2 are *type 1 contextually equivalent* whenever $cn_1 \simeq_1 cn_2$, i.e., whenever $cn_1\ \mathcal{R}\ cn_2$ for some type 1 witness relation $\mathcal{R}$.

2.5 Network-Aware Semantics

The type 1 semantics for μABS is quite abstract, and does not account for several issues which must be faced by an actual implementation, in particular if the goal is high performance and scalability. For instance, the type 1 semantics has no concept of proximity or name space. Any two objects, regardless of their "location", can, without any overhead or search, communicate via message passing in two steps, using the rules CALL-SEND and CALL-RCV. Instead, we want a semantics that is *network aware* in the sense that it brings out proximity and location without unduly constraining the model, e.g., to a particular naming discipline, or to a centralized name or location lookup service.

Our proposal is to execute μABS objects on a network of nodes in a fully decentralized and lock-free manner, where the only means of communication or synchronization is by asynchronous message passing along edges connecting nodes, each edge having an associated directional, buffered communication channel. In this section, we accordingly introduce a refinement of the standard semantics, a network-aware, type 2 semantics, which adds explicit network components to the type 1 semantics. The key idea is to use location independent routing, as explained in Section 2.1. Hence, nodes are equipped with explicit routing information, allowing messages to be addressed to specific receiving objects, rather than their last known host, from which they may have migrated.

The type 2 runtime syntax is presented in Table 2.3. We reuse symbols from the type 1 runtime syntax and use indices to disambiguate, and apply the same syntactical conventions. In particular, we continue to assume the commutativity and associativity properties of configuration juxtaposition, now with the empty list of containers as unit.

$u \in NID$			Node identifier
$t \in RTable$	$=$	$OID \to (NID \times \omega)_\perp$	Routing table
$q \in Q$	$=$	Msg^*	Message queue
$obj \in Obj_2$	$::=$	$\mathsf{o}(o, a, u, q_{in}, q_{out})$	Object
$nd \in Nd$	$::=$	$\mathsf{n}(u, t)$	Network node
$lnk \in Lnk$	$::=$	$\mathsf{l}(u, q, u')$	Network link
$ct \in Ct_2$	$::=$	$tsk \mid obj \mid nd \mid lnk$	Runtime container
$cn \in Cn_2$	$::=$	$ct_1 \ldots ct_n$	Configuration
$msg \in Msg$	$::=$	$\mathsf{table}(t) \mid \mathsf{object}(cn)$ $\mid \mathsf{call}(o, m, \overline{v})$	Message

Table 2.3: μABS type 2 runtime syntax

The network-aware runtime state is still a configuration cn, but most of the containers and their structure are different. We introduce two new types of containers to reflect the underlying network graph, namely *nodes* and *links*. A node container $\mathsf{n}(u, t)$ has a primitive *node identifier* $u \in NID$ with t an associated routing table.

Node identifiers (NIDs) take the place of IP addresses in the usual IP infrastructure. Nodes are connected by directed links of the form $\mathsf{l}(u, q, u')$, where $u \in \mathit{NID}$ is the source NID, $u' \in \mathit{NID}$ is the sink NID, and $q \in Q$ is the associated FIFO message queue. The node and link containers in a configuration cn induce a network graph $\texttt{graph}(cn)$, which contains a vertex u for each node container, and an edge (u, u') for each link container. The type 2 semantics given below does not allow identifiers in node or link containers to be changed, so in the context of any given transition (or, execution), the network graph remains constant. Note that there is no a priori guarantee that $\texttt{graph}(cn)$ is well-formed; for the remainder of the paper we therefore implicitly impose some well-formedness constraints on configurations concerning their network graphs.

Definition 2.5.1 (Network Graph Well-formedness)**.** A configuration cn has a *well-formed network graph* if it satisfies:

1. *Vertex Existence*: $\mathsf{n}(u, t) \preceq cn$ for some u and t.
2. *Edge Endpoint Existence*: If we have $\mathsf{l}(u_1, q, u_2) \preceq cn$, then $\mathsf{n}(u_1, t_1) \preceq cn$ and $\mathsf{n}(u_2, t_2) \preceq cn$.
3. *Unique Vertices*: If $\mathsf{n}(u_1, t_1)$ and $\mathsf{n}(u_2, t_2)$ are distinct occurrences in cn, then $u_1 \neq u_2$.
4. *Unique Edges*: If $\mathsf{l}(u_1, q_1, u_1')$ and $\mathsf{l}(u_2, q_2, u_2')$ are distinct occurrences in cn, then $u_1 \neq u_2$, or $u_1' \neq u_2'$, or both.
5. *Reflexivity*: If $\mathsf{n}(u, t) \preceq cn$, then $\mathsf{l}(u, q, u) \preceq cn$.
6. *Symmetry*: If we have $\mathsf{n}(u_1, t_1) \preceq cn$, $\mathsf{n}(u_2, t_2) \preceq cn$, and $\mathsf{l}(u_1, q, u_2) \preceq cn$, then $\mathsf{l}(u_2, q', u_1) \preceq cn$.
7. *Connectedness*: If $\mathsf{n}(u_1, t_1) \preceq cn$ and $\mathsf{n}(u_2, t_2) \preceq cn$, then there is a path from u_1 to u_2 in $\texttt{graph}(cn)$.

For routing, we assume a rudimentary Bellman-Ford d.v. routing discipline [193]. More elaborate and practical routing schemes exist that are better equipped for, e.g., disconnected operation, and with better combinations of scalability and stretch. However, the d.v. scheme is sufficient for our purposes. Consequently, a *routing table* t is a partial function associating to the OID o a pair $t(o) = (u, n)$, where n is the minimum number of hops believed by t to be needed to reach the node hosting object o from the current node, and where u is the NID of the next hop destination.

Next hop lookup is performed by the operation $\texttt{nxt}$, defined by $\texttt{nxt}(o, t) = \pi_1(t(o))$. The operation $\texttt{reg}$ returns the routing table t', obtained by registering a given object identifier o with distance n at the current node u of t; it is defined by

$$\texttt{reg}(o, u, t, n)(o') = \begin{cases} (u, n) & \text{if } o = o' \\ t(o') & \text{otherwise.} \end{cases}$$

The operation `upd` updates a table to incorporate the routing table belonging to a (neighboring) node, with $\mathtt{upd}(t, u, t')(o)$ defined by

$$\begin{array}{ll} \bot & \text{if } o \notin \mathtt{dom}(t) \cup \mathtt{dom}(t') \\ t(o) & \text{else, if } o \notin \mathtt{dom}(t') \\ (u, \pi_2(t'(o)) + 1) & \text{else, if } o \notin \mathtt{dom}(t) \text{ or } \pi_1(t'(o)) = u \\ (u, \pi_2(t'(o)) + 1) & \text{else, if } \pi_2(t'(o)) + 1 < \pi_2(t(o)) \\ t(o) & \text{otherwise.} \end{array}$$

The operations for FIFO message queues are standard: $\mathtt{enq}(msg, q)$ enqueues the message *msg* onto the tail of q, $\mathtt{hd}(q)$ returns the head of q, and $\mathtt{deq}(q)$ returns the tail of the q, i.e. q with $\mathtt{hd}(q)$ removed. If q is empty, i.e., $q = \varepsilon$, then $\mathtt{hd}(q)$ and $\mathtt{deq}(q)$ are both undefined.

Messages in queues have one of three forms:

- $\mathsf{call}(o, m, \overline{v})$ is a call message addressed to object o, for method m, with arguments $\overline{v}$;
- $\mathsf{table}(t)$ is a routing table update message, with the origin NID implicit, as the message is dequeued from a link queue with explicit source NID;
- $\mathsf{object}(cn)$ is an object migration message, where cn is an *object closure*, as explained below.

We write $\mathsf{m}(cn)$ for the multiset $\{msg \mid msg \preceq cn\}$. Call messages are said to be *object bound*, and table and object messages are said to be *node bound*. We define $\mathtt{dst}(msg)$, the *destination* of *msg*, to be o for call messages, and $\bot$ for the two other message forms.

In the type 2 semantics, object containers are now attached to a node u and have the shape $\mathsf{o}(o, a, u, q_{in}, q_{out})$, where $o \in \mathit{OID}$ and $a \in \mathit{OEnv}$ as before, and q_{in} and q_{out} is an ingoing and an outgoing FIFO message queue, respectively. This buffering of messages in object state is not essential, since messages are already buffered at the link level, but allows for a more straightforward formalization of object behavior. In contrast to many actor languages, e.g., Erlang, which use only ingoing queues or *mailboxes*, we find it more convenient to use an outgoing queue as well, although this is mainly a matter of taste. Tasks are unchanged from the type 1 semantics.

For an object message $\mathsf{object}(cn)$ to be valid, the configuration cn needs to be an *object closure* which wraps an object container with all its tasks. For example, if the object o has precisely n tasks, its closure has the form

$$cn = \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l_1, s_1)\ \ldots\ \mathsf{t}(o, l_n, s_n).$$

Note that containers inside object closures are included in the subterm relation $\preceq$ for configurations where the object message resides. To handle closures in the semantics, we define three auxiliary operations:

- `clo`(cn, o) is the closure of object o with respect to cn, defined as the multiset of all type 2 containers in cn which are of the form $\mathtt{o}(o', a', u', q'_{in}, q'_{out})$ or $\mathtt{t}(o', l', s')$, and are such that $o' = o$;

- `oidof`(cn) is a partial function returning o if all type 2 containers in cn are object or task containers with OID o;

- `place`(cn, u) places all object containers in the configuration cn at the node u, i.e., cn and `place`(cn, u) are identical, except that each object container $\mathtt{o}(o', a', u', q'_{in}, q'_{out})$ in cn is replaced by an object container $\mathtt{o}(o', a', u, q'_{in}, q'_{out})$ in `place`(cn, u).

Example 2.5.2. Figure 2.2 shows a pictorial representation of a network-aware μABS runtime configuration, with two nodes u_0 and u_1, an object o_0 located at u_0 with tasks $\mathtt{t}_0$ and $\mathtt{t}_1$, and an object o_1 located at u_1 with a task $\mathtt{t}_2$. Note that routing table information on both nodes is accurate and complete.

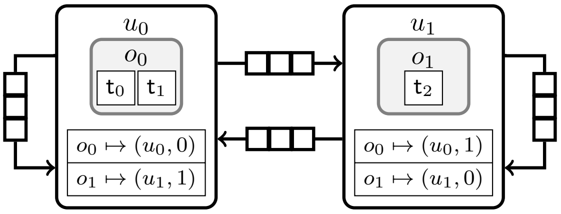


Figure 2.2: μABS network-aware runtime configuration

An important distinction between the reference semantics and the network-aware semantics is the absence of binding. For the standard semantics, name binding plays a key role in avoiding clashes between locally generated names. However, in a language with NIDs, this device is no longer needed, since globally unique names can be guaranteed by augmenting names with their generating NID. Since all name generation takes place in the context of a given NID, we can simply assume an operation `newo`(u) that returns a new OID which is globally fresh for the "current configuration". Another important point to note is that all transitions in the type 2 semantics are *fully local*, in the sense that all operations applied, and all conditions determining whether or not a transition is enabled, can be determined by inspecting only one node and, possibly, the head of incoming link queues, or by enqueuing messages to the tail of an outgoing queue.

The reduction rules in Figure 2.1, except CTXT-2, CALL-SEND, CALL-RCV, and NEW, are transferred to the type 2 setting with minor modifications—more precisely, WLOCAL, WFIELD, IF-TRUE, IF-FALSE, WHILE-TRUE, and WHILE-FALSE are changed in the obvious way to accommodate the new runtime shape of objects.

CTXT-1: If $cn_1 \to cn_2$, then $cn \vdash cn_1 \to cn_2$

WLOCAL-2: If $x \in \mathtt{dom}(l)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, x = e; s) \to \mathsf{t}(o, l[v/x], s)$

WFIELD-2: If $x \in \mathtt{dom}(a)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$

SKIP: $\mathsf{t}(o, l, \mathbf{skip}) \to 0$

IF-TRUE-2: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, \mathbf{if}\ e\ \{s_1\}\ \mathbf{else}\ \{s_2\}; s) \to \mathsf{t}(o, l, s_1; s)$

IF-FALSE-2: If $[\![e]\!]_{(a,l)} = 0$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, \mathbf{if}\ e\ \{s_1\}\ \mathbf{else}\ \{s_2\}; s) \to \mathsf{t}(o, l, s_2; s)$

WHILE-TRUE-2: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\ \{s_1\}; s) \to \mathsf{t}(o, l, s_1; \mathbf{while}\ e\ \{s_1\}; s)$

WHILE-FALSE-2: If $[\![e]\!]_{(a,l)} = 0$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\ \{s_1\}; s) \to \mathsf{t}(o, l, s)$

Figure 2.3: μABS type 2 reduction rules, part 1

The result is the set of rules in Figure 2.3. The remaining reduction rules are presented in Figure 2.4.

T-SEND and T-RCV are concerned with the exchange of routing tables, which only takes place between distinct adjacent nodes. MSG-SEND, MSG-RCV, and MSG-ROUTE are used to manage message passing, i.e., reading a message from a link queue and transferring it to the appropriate object in-queue, and, dually, reading a message from an out-queue and transferring it to the attached link queue. Finally, messages are routed to the next link, if the destination object does not reside at the current node. In MSG-RCV, note that the receiving node is not required to be present. However, its existence follows from the well-formedness conditions of the network graph.

MSG-DELAY-1, MSG-DELAY-2, and MSG-DELAY-3 are used to handle the cases where routing tables have not yet stabilized, or a message is simply unroutable. For instance, it may happen that updates to the routing tables have not yet caught up with object migration. In this case, a message may enter an out-queue without the hosting node's routing table having information about the message's destination (MSG-DELAY-2). Another case is when a node receives a message on a link without knowing where to forward it (MSG-DELAY-1). This situation is particularly problematic, since a blocked message may prevent routing table updates from reaching the hosting node, thus causing a deadlock. The solution we propose, which is implicit in the rules, is to use the network self-loop as a buffer for temporarily unroutable messages. MSG-DELAY-3 allows messages on this link to be shuffled.

T-SEND: If $u \neq u'$, then $\mathsf{n}(u,t) \vdash \mathsf{l}(u,q,u') \to \mathsf{l}(u, \mathsf{enq}(\mathsf{table}(t), q), u')$

T-RCV: If $\mathtt{hd}(q) = \mathsf{table}(t')$, then
$\mathsf{l}(u',q,u)\ \mathsf{n}(u,t) \to \mathsf{l}(u', \mathsf{deq}(q), u)\ \mathsf{n}(u, \mathsf{upd}(t,u',t'))$

MSG-SEND: If $\mathtt{hd}(q_{out}) = msg$, $\mathtt{dst}(msg) = o'$, $\mathtt{nxt}(o',t) = u'$, then
$\mathsf{n}(u,t) \vdash \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{l}(u,q,u') \to$
$\mathsf{o}(o,a,u,q_{in},\mathsf{deq}(q_{out}))\ \mathsf{l}(u,\mathsf{enq}(msg,q),u')$

MSG-RCV: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, then
$\mathsf{l}(u',q,u)\ \mathsf{o}(o,a,u,q_{in},q_{out}) \to \mathsf{l}(u',\mathsf{deq}(q),u)\ \mathsf{o}(o,a,u,\mathsf{enq}(msg,q_{in}),q_{out})$

MSG-ROUTE: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, $\mathtt{nxt}(o,t) = u''$, $u'' \neq u$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u',q,u)\ \mathsf{l}(u,q',u'') \to \mathsf{l}(u',\mathsf{deq}(q),u)\ \mathsf{l}(u,\mathsf{enq}(msg,q'),u'')$

MSG-DELAY-1: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, $\mathtt{nxt}(o,t)\uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u',q,u)\ \mathsf{l}(u,q',u) \to \mathsf{l}(u',\mathsf{deq}(q),u)\ \mathsf{l}(u,\mathsf{enq}(msg,q'),u)$

MSG-DELAY-2: If $\mathtt{hd}(q_{out}) = msg$, $\mathtt{dst}(msg) = o'$, $\mathtt{nxt}(o',t)\uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{l}(u,q,u) \to$
$\mathsf{o}(o,a,u,q_{in},\mathsf{deq}(q_{out}))\ \mathsf{l}(u,\mathsf{enq}(msg,q),u)$

MSG-DELAY-3: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, $\mathtt{nxt}(o,t)\uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u,q,u) \to \mathsf{l}(u,\mathsf{enq}(msg,\mathsf{deq}(q)),u)$

CALL-SEND-2: Let $o' = [\![e_1]\!]_{(a,l)}$, $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$ in
$\mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,e_1!m(\overline{e_2});s) \to$
$\mathsf{o}(o,a,u,q_{in},\mathsf{enq}(\mathsf{call}(o',m,\overline{v}),q_{out}))\ \mathsf{t}(o,l,s)$

CALL-RCV-2: If $\mathtt{hd}(q_{in}) = \mathsf{call}(o,m,\overline{v})$ then
let $l = \mathtt{locals}(o,m,\overline{v})$, $s = \mathtt{body}(o,m)$ in
$\mathsf{o}(o,a,u,q_{in},q_{out}) \to \mathsf{o}(o,a,u,\mathsf{deq}(q_{in}),q_{out})\ \mathsf{t}(o,l,s)$

NEW-2: Let $o' = \mathtt{newo}(u)$, $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \mathtt{init}(C,\overline{v},o')$ in
$\mathsf{o}(o,a,u,q_{in},q_{out}) \vdash \mathsf{t}(o,l,x = \textbf{new}\ C(\overline{e});s) \to$
$\mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a',u,\varepsilon,\varepsilon)$

OBJ-REG: $\mathsf{o}(o,a,u,q_{in},q_{out}) \vdash \mathsf{n}(u,t) \to \mathsf{n}(u,\mathtt{reg}(o,u,t,0))$

OBJ-SEND: If $u \neq u'$, then let $cn' = \mathtt{clo}(cn,o)$ in
$\mathsf{n}(u,t)\ \mathsf{l}(u,q,u')\ cn \to \mathsf{n}(u,\mathtt{reg}(o,u',t,1))\ \mathsf{l}(u,\mathsf{enq}(\mathsf{object}(cn'),q),u')\ (cn - cn')$

OBJ-RCV: If $\mathtt{hd}(q) = \mathsf{object}(cn)$, then
$\mathsf{l}(u',q,u)\ \mathsf{n}(u,t) \to \mathsf{l}(u',\mathsf{deq}(q),u)\ \mathsf{n}(u,\mathtt{reg}(\mathtt{oidof}(cn),u,t,0))\ \mathtt{place}(cn,u)$

Figure 2.4: μABS type 2 reduction rules, part 2

CALL-SEND-2 and CALL-RCV-2 produce and consume call messages in an obvious way, corresponding closely to their type 1 counterparts. NEW-2 handles object creation. Note that a new object does not automatically get registered in the hosting node's routing table.

OBJ-REG registers an object in the routing table at the node on which it is located. The final two rules concern object migration. Of these, OBJ-SEND is a *global* rule in that it is not allowed to be used in subsequent applications of the CTXT-1 rule. In this way, we can guarantee that only complete object closures are migrated. The rule can still be implemented using local operations, since an object and all its tasks are implicitly assumed to be co-located on a node. In OBJ-SEND, $cn - cn'$ is multiset difference.

The reduction rules are formulated with the main goal of making it straightforward to reason about them. Hence, the rules can be optimized in several ways to exhibit behavior more suitable for implementation. For instance, object self-calls will generally be routed through the "network interface", i.e., the hosting node's self-loop. This is not necessary. It would be possible to add a rule to directly spawn a handling task from a self call without affecting the results of the paper.

As for the reference semantics, some basic sanity properties of the network-aware semantics can be established.

Proposition 2.5.3. *Suppose $cn \to cn'$. Then, the following holds:*

1. *If $\mathsf{n}(u,t) \preceq cn$, then $\mathsf{n}(u,t') \preceq cn'$ for some t'.*
2. *If $\mathsf{l}(u,q,u') \preceq cn$, then $\mathsf{l}(u,q',u') \preceq cn'$ for some q'.*
3. *If $obj = \mathsf{o}(o,a,u,q_{in},q_{out}) \preceq cn$, then there is an object container $obj' = \mathsf{o}(o',a',u',q'_{in},q'_{out}) \preceq cn'$ (the* derivative *of obj in cn') such that $o' = o$ and for all x, if $a(x)\downarrow$, then $a'(x)\downarrow$.*

Initial configurations in the network-aware semantics are now parameterized on a network graph, but are otherwise similar to their network-oblivious counterparts.

Definition 2.5.4 (Type 2 Initial Configuration)**.** Consider a program $\overline{CL}\ \{\overline{x}, s\}$. Assume a reserved OID o_{main}. A *type 2 initial configuration cn_{init}* for the program then has the shape

$$cn_{graph}\ \mathsf{o}(o_{main}, \bot, u_{init}, \varepsilon, \varepsilon)\ \mathsf{t}(o_{main}, l_{init}, s)$$

where

- $\bot$ and l_{init} are as for type 1 initial configurations,
- cn_{graph} is a configuration consisting only of nodes and links with empty queues, inducing a well-formed graph,
- cn_{graph} contains a node $\mathsf{n}(u_{init}, t_{init})$,
- $t_{init}(o_{main}) = (u_{init}, 0)$, and $t_{init}(o) = \bot$ for $o \neq o_{main}$,
- $t(o) = \bot$ for all $t \neq t_{init}$ and OIDs o.

The previous well-formedness conditions need to be augmented for the type 2 semantics, due to the addition of new containers.

Definition 2.5.5 (Type 2 Well-formedness)**.** A type 2 configuration cn is *type 2 well-formed* (WF2) if cn satisfies:

1. *OID Uniqueness*: If $\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1})$ and $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2})$ are distinct object container occurrences in cn, then $o_1 \neq o_2$.

2. *Task-Object Existence*: If we have $\mathsf{t}(o, l, s) \preceq cn$, then $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ for some a, u, q_{in}, q_{out}.

3. *Object-Node Existence*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, then $\mathsf{n}(u, t) \preceq cn$ for some t.

4. *Buffer Cleanliness*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ and additionally $msg \preceq q_{in}$ or $msg \preceq q_{out}$, then msg is object bound. Also, if $msg \preceq q_{in}$, then $\mathtt{dst}(msg) = o$.

5. *Local Routing Consistency*: If we have $\mathsf{n}(u, t) \preceq cn$ and $\pi_1(\mathtt{nxt}(o, t)) = u'$, then there is a link $\mathsf{l}(u, q, u') \preceq cn$.

6. *External OID*: $ext \notin OID(cn)$, and if $\mathsf{n}(u, t) \preceq cn$, then $ext \notin \mathtt{dom}(t)$.

For condition 4 in Definition 2.5.5, observe that only object-bound messages (meant for in-queues, appropriately addressed) enter the object queues. This condition is needed to prevent the deadlocks that arise when an in- or out-queue contains messages of the wrong type. Condition 6 in Definition 2.5.5 ensures that messages to ext are only transported to self-loop links, where they remain. We note that type 2 well-formedness still holds for initial configurations and is preserved under reduction.

Proposition 2.5.6 (WF2 Preservation)**.** *Let cn be a configuration. Then, the following holds:*

1. *If cn is a type 2 initial configuration, then cn is WF2.*

2. *If cn is WF2 and $cn \to cn'$, then cn' is WF2.*

Example 2.5.7. Figure 2.5 illustrates a fragment of a network-aware execution of a μABS program, with node self-loop queues omitted. The network consists of four nodes and two objects. Initially, in Figure 2.5(a), the objects o_0 and o_1 are located on u_0 and u_3, respectively, with routing tables stabilized; o_0 has a task with a call statement for method m in o_1 with argument 7. In Figure 2.5(b), a call message has been dispatched from u_0 to node u_1, while the object o_1, represented as obj_1, is in transit from u_3 to u_2. Note that routing tables are now in an unstable state. In Figure 2.5(c), the o_1 has arrived at u_2, and through the accurate route for o_1 at u_3, the call message has been sent in the direction of u_2. Finally, in Figure 2.5(d), all routing tables are stable once again after o_1 arrives at u_2, routing tables are exchanged between neighboring nodes, and the call message has been delivered to o_1, generating a task with statement s. If the objects subsequently remain at their locations, messages between them will now take the direct route from u_0 to u_2.

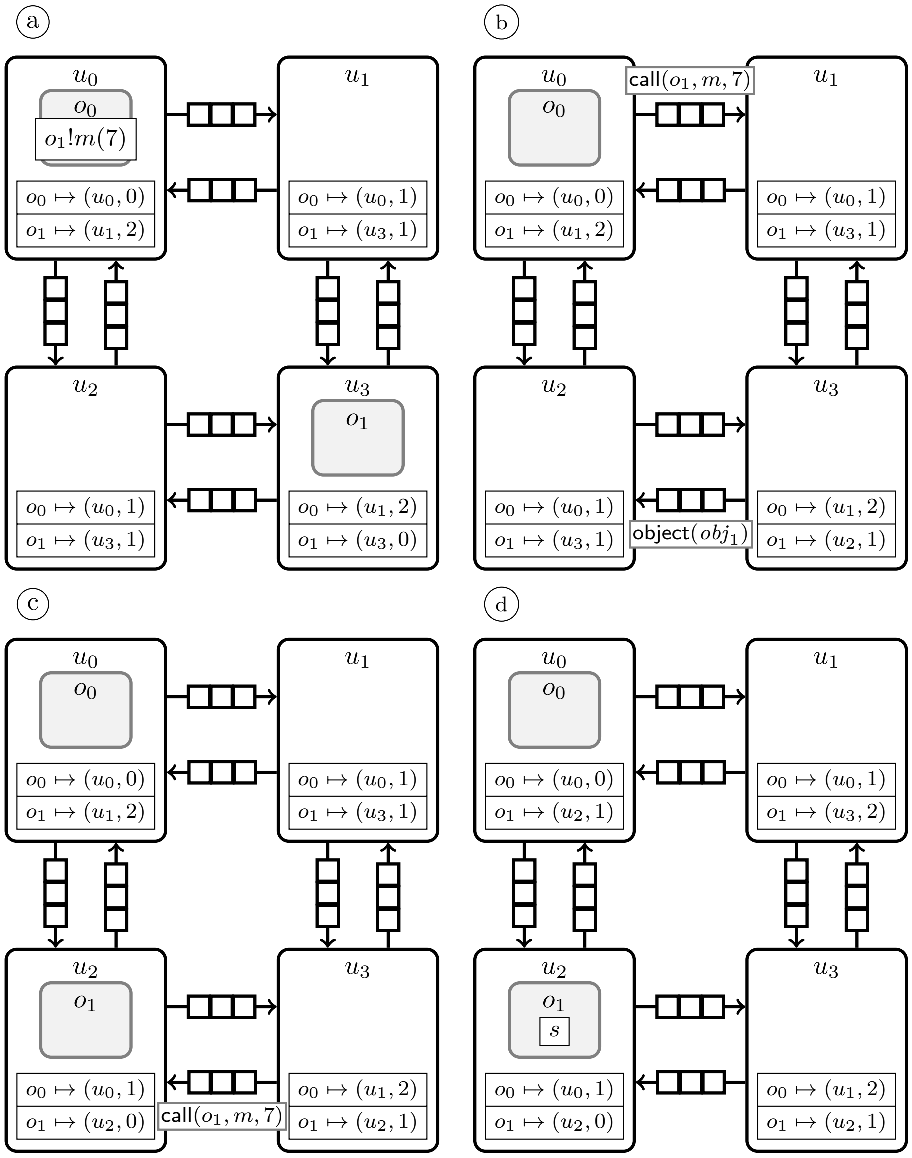


Figure 2.5: Fragment of a μABS network-aware execution

We next adapt the notion of contextual equivalence to the type 2 setting. The only real difficulty is to define the type 2 correlate of the observation predicate. We take the point of view that an observation $obs = ext!m(\overline{v})$ is enabled at a configuration cn if a corresponding call message $\mathsf{call}(ext, m, \overline{v})$ is located at the head of one of the self-loop queues in cn. More precisely, the type 2 observability predicate is $cn \downarrow obs$, holding just in case $cn = cn'\ \mathsf{l}(u, q, u)$ for some cn', u, and q, and additionally $\mathtt{hd}(q)$ is defined and equal to $\mathsf{call}(ext, m, \overline{v})$.

For context closure, a *context* is any configuration cn containing only object and task containers. Thus, contexts do not affect the underlying network graph. This definition is used, since, firstly, it is objects and tasks that induce computational behavior, and secondly, allowing contexts to augment the underlying graph by adding new nodes and links requires a much more complex account of network composition and well-formedness that we prefer to leave to future work.

With the observation predicate set up, the weak observation predicate is derived as for type 1 configurations, and, similar to in Definition 2.4.5, we define a *type 2 witness relation* as a relation that satisfies reduction closure and barb preservation, with context closure defined as follows: if $cn_1\ \mathcal{R}\ cn_2$, cn is a context, and $cn_1\ cn$ is WF2, then $cn_2\ cn$ is WF2, and $cn_1\ cn\ \mathcal{R}\ cn_2\ cn$.

Definition 2.5.8 (Type 2 Contextual Equivalence)**.** Let $cn_1 \simeq_2 cn_2$ whenever $cn_1\ \mathcal{R}\ cn_2$ for a type 2 witness relation $\mathcal{R}$.

2.6 Normal Forms

The goal is to prove the type 1 behavior is preserved in the type 2 semantics under contextual equivalence. The key to the proof is a normal form lemma for the type 2 semantics saying, roughly, that any well-formed type 2 configuration can be rewritten, using a subset of the rules as detailed below, into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, and where all objects have been moved to a single node. We prove this lemma in two steps. First, we prove a stabilization result: that non-loop links can be emptied of messages and routing tables normalized to induce messaging paths with unit stretch. This allows the second normalization step to empty also object queues and migrate all objects to a single node. Once this is done, we can prove correctness by exhibiting a map representing each type 1 configuration as a canonical type 2 configuration, using normalization to help prove reduction closure and context closure in both directions.

In a configuration cn, we call a link $\mathsf{l}(u, q, u') \preceq cn$ *proper* whenever $u \neq u'$, and say that a message msg is *routable* whenever $\mathtt{dst}(msg) \in OID(cn)$ and *unroutable* otherwise. We first show that each configuration can be rewritten using the transition rules into a form for which routing is stable and all link queues are empty, except for object-bound, unroutable messages.

Definition 2.6.1 (Stable Routing)**.** Let *cn* be a type 2 configuration. We say that *cn* has *stable routing*, if, for all $\mathsf{n}(u, t) \preceq cn$ and $\mathsf{o}(o, a, u', q_{in}, q_{out}) \preceq cn$, if $\mathtt{nxt}(o, t) = u''$, then there is a minimum-length path from u to u' in $\mathtt{graph}(cn)$ that traverses u''.

Definition 2.6.2 (External Link Messages)**.** Let *cn* be a type 2 configuration. We say that *cn* has *external link messages*, if $\mathsf{l}(u, q, u') \preceq cn$, and $msg \preceq q$ implies $u = u'$ and that *msg* is object bound and unroutable.

The strategy for performing the rewriting is to first empty link queues as far as possible as we simultaneously exchange routing tables to converge to a configuration with stable routing. This first stage is accomplished using Algorithm 1, displayed in Listing 2.2, where we hide uses of CTXT-1 to allow the transition rules to be applied to arbitrary containers.

Algorithm 1: Stabilize routing and process link messages

Input: A WF2 configuration *cn*
Output: A configuration in stable form, reachable from *cn*

```
repeat
  use OBJ-REG on each object not in transit ;
  use T-SEND on each proper link to broadcast routing tables
  from all nodes to their neighboring nodes ;
  repeat
    use T-RCV to dequeue one message on a link
  until T-RCV can no longer be used ;
  once for each link, if possible, use MSG-RCV, MSG-ROUTE,
  MSG-DELAY-1, or OBJ-RCV, or otherwise, use MSG-DELAY-3
  on self-loop links with external messages at the queue head
until routing has stabilized and there are only external link messages
```

Listing 2.2: Algorithm 1

Proposition 2.6.3. *Algorithm 1 terminates.*

Write $\mathcal{A}_1(cn) \rightsquigarrow cn'$ if cn' is a possible result of applying Algorithm 1 to *cn*. The resulting configuration is almost unique, but not quite, since routing may stabilize in different ways. We make the notion of stabilization precise using some auxiliary functions:

- $\mathsf{t}(cn)$ is the multiset of all tasks in *cn*;
- $\mathsf{o}_1(cn)$ is the object multiset where, if $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, there is a corresponding container $\mathsf{o}(o, a, u', q'_{in}, q_{out})$, such that the NID u' has been adjusted to that of the receiving node if the object was in transit from u to

u' in cn, or otherwise $u' = u$, and additionally, all messages in link queues in cn such that $\texttt{dst}(msg) = o$ have been enqueued in some fixed order in q_{in} to produce q'_{in};

- $\mathsf{m}_1(cn)$ is the multiset of both external messages and routable messages in cn.

Define the relation $\cong_1$ (different from $\simeq_1$) to hold between multisets of type 2 object containers when there is a one-to-one mapping where containers only differ in how in-queue messages are ordered, if at all.

Definition 2.6.4 (Stable Form)**.** The configuration cn is in *stable form* if it is WF2, has stable routing, $\mathsf{o}(cn) \cong_1 \mathsf{o}_1(cn)$, and $\mathsf{m}(cn) = \mathsf{m}_1(cn)$.

Proposition 2.6.5. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then:*

1. $cn \rightarrow^* cn'$,
2. *cn' is in stable form,*
3. $\texttt{graph}(cn') = \texttt{graph}(cn)$,
4. $\mathsf{t}(cn') = \mathsf{t}(cn)$,
5. $\mathsf{o}(cn') \cong_1 \mathsf{o}_1(cn)$, *and*
6. $\mathsf{m}(cn') = \mathsf{m}_1(cn)$.

We can now define equivalence of configurations up to stabilization, and show that it is contained in $\simeq_2$.

Definition 2.6.6 ($\equiv_1$)**.** Let $cn_1 \ \mathcal{R}_1 \ cn_2$ whenever we have

1. $\texttt{graph}(cn_1) = \texttt{graph}(cn_2)$,
2. $\mathsf{t}(cn_1) = \mathsf{t}(cn_2)$,
3. $\mathsf{o}(cn_1) \cong_1 \mathsf{o}(cn_2)$, and
4. $\mathsf{m}(cn_1) = \mathsf{m}(cn_2)$.

We say that $cn_1 \equiv_1 cn_2$ if cn_1 and cn_2 are both WF2, and there are configurations cn'_1 and cn'_2 such that

$$\mathcal{A}_1(cn_1) \rightsquigarrow cn'_1 \ \mathcal{R}_1 \ cn'_2 \leftsquigarrow \mathcal{A}_1(cn_2).$$

Corollary 2.6.7. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then $cn \equiv_1 cn'$.*

Lemma 2.6.8. *$\equiv_1$ is reduction closed.*

Lemma 2.6.9. *$\equiv_1$ is context closed.*

Proposition 2.6.10. $\equiv_1$ *is a type 2 witness relation.*

Corollary 2.6.11. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then $cn \simeq_2 cn'$.*

The second procedure, Algorithm 2, shown in Listing 2.3, empties object queues and migrates object closures to a "central" node. Initially, the node u is chosen,

Algorithm 2: Normalization

Input: A WF2 configuration cn
Output: A configuration in normal form, reachable from cn

```
fix a NID $u$ for a node in $cn$ ;
run Algorithm 1 ;
repeat
  while some object queue is nonempty
    use MSG-SEND, MSG-DELAY-2, or CALL-RCV-2 to
    dequeue one message from each nonempty object queue ;
  end ;
  while an object $o$ exists not located at $u$
    use OBJ-SEND to send $o$ towards $u$
  end ;
  run Algorithm 1
until all objects are located at $u$, all object queues are empty, and there are only
external link messages
```

Listing 2.3: Algorithm 2

towards which all objects will migrate during normalization. Normalization is then performed in cycles, with a cycle starting and ending in a stable form. In a cycle, object in- and out-queues are first emptied. Then, objects are migrated one step towards u. Routing is not needed to perform such steps; it is sufficient to know that migration towards u is possible, which it is by the well-formedness of the network graph.

Proposition 2.6.12. *Algorithm 2 terminates.*

Write $\mathcal{A}_2(cn) \rightsquigarrow cn'$ if cn' is a possible result of applying Algorithm 2 to cn. As before, we define some auxiliary functions:

- $\mathsf{t}_2(cn)$ is the multiset of method containers $tsk = \mathsf{t}(o, l, s)$ such that either $tsk \preceq cn$, or there is a routable message $\mathsf{call}(o, m, \overline{v})$ in transit such that $l = \mathtt{locals}(o, m, \overline{v})$, and $s = \mathtt{body}(o, m)$;
- $\mathsf{o}_2(cn)$ is the multiset of object containers $\mathsf{o}(o, a, u, \varepsilon, \varepsilon)$ such that there is an object container $\mathsf{o}(o, a, u', q_{in}, q_{out}) \preceq cn$;
- $\mathsf{m}_2(cn)$ is the multiset of external messages in queues in cn.

Using these functions, we can describe the effects of normalization, i.e., running Algorithm 2 on a configuration, as made precise in the following definition and proposition.

Definition 2.6.13 (Normal Form)**.** The configuration cn is in *normal form* if it is WF2, has stable routing and external link messages, $\mathsf{t}(cn) = \mathsf{t}_2(cn)$, $\mathsf{o}(cn) = \mathsf{o}_2(cn)$, and $\mathsf{m}(cn) = \mathsf{m}_2(cn)$.

Proposition 2.6.14. *If cn is WF2 and $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then:*

1. $cn \rightarrow^* cn'$,
2. *cn' is in normal form,*
3. $\mathtt{graph}(cn') = \mathtt{graph}(cn)$,
4. $\mathsf{t}(cn') = \mathsf{t}_2(cn)$,
5. $\mathsf{o}(cn') = \mathsf{o}_2(cn)$, *and*
6. $\mathsf{m}(cn') = \mathsf{m}_2(cn)$.

As for stabilization, we define an equivalence up to normalization, contained in $\simeq_2$, but more extensive than $\equiv_1$. This equivalence is key to our correctness argument.

Definition 2.6.15 ($\equiv_2$)**.** Let $cn_1 \; \mathcal{R}_2 \; cn_2$ whenever we have

1. $\mathtt{graph}(cn_1) = \mathtt{graph}(cn_2)$,
2. $\mathsf{t}(cn_1) = \mathsf{t}(cn_2)$,
3. $\mathsf{o}(cn_1) = \mathsf{o}(cn_2)$, and
4. $\mathsf{m}(cn_1) = \mathsf{m}(cn_2)$.

We say that $cn_1 \equiv_2 cn_2$ if cn_1 and cn_2 are both WF2, and there are configurations cn_1' and cn_2' such that

$$\mathcal{A}_2(cn_1) \rightsquigarrow cn_1' \; \mathcal{R}_2 \; cn_2' \leftsquigarrow \mathcal{A}_2(cn_2).$$

Corollary 2.6.16. $\equiv_1 \subseteq \equiv_2$.

Corollary 2.6.17. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \equiv_2 cn'$.*

Lemma 2.6.18. *$\equiv_2$ is reduction closed.*

Lemma 2.6.19. *$\equiv_2$ is context closed.*

Proposition 2.6.20. *$\equiv_2$ is a type 2 witness relation.*

Corollary 2.6.21. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \simeq_2 cn'$.*

2.7 Correctness

To show soundness and full abstraction for the network-aware semantics, we define a mapping net taking type 1 configurations to their close-to-normal-form type 2 counterparts. We first fix an underlying well-formed network graph represented as a type 2 configuration cn_{graph} and a distinguished NID u_0 for a node in this graph. Thus, cn_{graph} consists only of node containers and link containers with empty queues. The routing tables of nodes are defined later. A first complication in defining a suitable representation map is that names in the type 1 semantics (which includes the binder bind) need to be related to names in the type 2 semantics, which does not include the binder, but on the other hand has the name generation operation newo. For *ext*, this is not a problem, but for other names, some form of name representation map is needed to connect the two types of names. Accordingly, we fix an injective *name representation map* rep, taking an object name o in the type 1 semantics to an object identifier $\texttt{rep}(o)$ in the type 2 semantics. We extend the name representation map rep to environments and arbitrary values:

- $\texttt{rep}(p) = p$, if $p \in PVal$
- $\texttt{rep}(l)(x) = \texttt{rep}(l(x))$ and $\texttt{rep}(a)(x) = \texttt{rep}(a(x))$
- $\texttt{rep}(a)(\textbf{self}) = \texttt{rep}(a(\textbf{self}))$ and $\texttt{rep}(ext) = ext$

Since we have left the nature of expressions unspecified, we need to additionally assume that rep commutes with the expression semantics, i.e., for all e, a, and l,

$$\texttt{rep}([\![e]\!]_{(a,l)}) = [\![e]\!]_{(\texttt{rep}(a),\texttt{rep}(l))}.$$

Now, the only remaining complication in defining the mapping net is that we need to deliver message-converted type 1 call containers into a message queue. This is done by the operation send, which puts call messages in the self-loop queue of u_0, where

$$\begin{aligned}&\texttt{send}(\mathsf{call}(o, m, \overline{v}), \mathsf{l}(u_0, q, u_0)\ cn) = \\ &\quad \mathsf{l}(u_0, \mathsf{enq}(\mathsf{call}(o, m, \overline{v}), q), u_0)\ cn.\end{aligned}$$

Given a name representation map rep, we can then define the type 2 representation of a type 1 configuration with a transformer, as follows:

- $\texttt{net}(cn_1\ cn_2, \texttt{rep}) = \texttt{net}(cn_1, \texttt{rep}) \circ \texttt{net}(cn_2, \texttt{rep})$
- $\texttt{net}(0, \texttt{rep})(cn) = cn$
- $\texttt{net}(\mathsf{t}(o, l, s), \texttt{rep})(cn) = \mathsf{t}(\texttt{rep}(o), \texttt{rep}(l), s)\ cn$
- $\texttt{net}(\mathsf{o}(o, a), \texttt{rep})(cn) = \mathsf{o}(\texttt{rep}(o), \texttt{rep}(a), u_0, \varepsilon, \varepsilon)\ cn$

- $\mathtt{net}(\mathsf{c}(o, m, \overline{v}), \mathtt{rep})(cn) =$
  $\mathsf{send}(\mathsf{call}(\mathtt{rep}(o), m, \mathtt{rep}(\overline{v})), cn)$

Hence, we represent WF1 configurations by first assuming some underlying network graph, and then mapping the containers individually to the type 2 level, resulting in a WF2 configuration. The initial routing table t_0 at u_0 is now defined to have all OIDs in the configuration registered as local, i.e.,

$$t_0 = \mathtt{reg}(\mathtt{rep}(o_1), u_0, \mathtt{reg}(\cdots, \mathtt{reg}(\mathtt{rep}(o_n), u_0, \perp))\cdots).$$

For nodes $\mathsf{n}(u, t)$ where $u \neq u_0$, we let t be determined by some stable routing, using Algorithm 1.

Definition 2.7.1 (Representation Map $\mathtt{net}$)**.** Let a network configuration cn_{graph} and a name representation map $\mathtt{rep}$ be given for a WF1 configuration $\mathsf{bind}\ \overline{o}.cn$ in standard form. Then, the *type 2 representation* of cn is defined as $\mathtt{net}(cn) = \mathtt{net}(cn, \mathtt{rep})(cn_{graph})$.

The most basic property that we expect to hold about $\mathtt{net}$ is that it produces a type 2 well-formed configuration when given a type 1 well-formed configuration.

Proposition 2.7.2. *If* $\mathsf{bind}\ \overline{o}.cn$ *is a WF1 configuration in standard form, then* $\mathtt{net}(cn)$ *is WF2.*

We now obtain a key lemma allowing us to relate transitions in the two semantics under normal form equivalence, and thus contextual equivalence, leading up to the main result.

Lemma 2.7.3. *Let* $\mathsf{bind}\ \overline{o}.cn$ *be a type 1 well-formed configuration in standard form. Then:*

1. *If* $\mathsf{bind}\ \overline{o}.cn \to \mathsf{bind}\ \overline{o}'.cn'$*, then for some* cn''*,* $\mathtt{net}(cn) \to^* cn''$ *and* $cn'' \equiv_2 \mathtt{net}(cn')$*.*
2. *If* $\mathtt{net}(cn) \to cn''$*, then for some* $\overline{o}'$ *and* cn'*,* $\mathsf{bind}\ \overline{o}.cn \to^* \mathsf{bind}\ \overline{o}'.cn'$ *and* $cn'' \equiv_2 \mathtt{net}(cn')$*.*

For both properties in Lemma 2.7.3, the argument is by case analysis on the possible rules applied in the assumed reduction step, using the aforementioned commutativity property of $\mathtt{rep}$ with the expression semantics where necessary to produce a desired configuration.

Given our configuration mapping $\mathtt{net}$, with a name representation map $\mathtt{rep}$, we now conflate our notions of type 1 and type 2 witness relation into a notion that includes relations between WF1 and WF2 configurations, leading to a generalized contextual equivalence, $\simeq$. For such a conflated witness relation $\mathcal{R}$, reduction closure and barb preservation, as in Definition 2.4.5, is straightforward to define. The main problem lies in defining the notion of context closure, which requires applying a context configuration to two different configuration types. Applying a type 1 context *cn* to a type 2 configuration involves faithfully transforming elements of *cn* to the type 2 level, by introducing, e.g., locations to objects, and turning call containers into messages. Conversely, applying a type 2 context to a type 1 configuration involves removing locations and queues, and turning messages into call containers.

More formally, suppose $\mathsf{bind}\ \overline{o}.cn_1\ \mathcal{R}\ cn_2$, with $\mathsf{bind}\ \overline{o}.cn_1$ WF1 and in standard form, and cn_2 WF2. Assume that, when we apply the type 1 context configuration cn to $\mathsf{bind}\ \overline{o}.cn_1$, we get the configuration $\mathsf{bind}\ \overline{o}'.cn_1\ cn'$ in standard form. We then apply the context to cn_2 by defining the result as $\mathtt{net}(cn', \mathtt{rep})(cn_2)$. Consequently, context closure requires that $\mathsf{bind}\ \overline{o}'.cn_1\ cn'\ \mathcal{R}\ \mathtt{net}(cn', \mathtt{rep})(cn_2)$ in this case. Conversely, suppose $cn_2\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{o}.cn_1$, again with cn_2 WF2, and $\mathsf{bind}\ \overline{o}.cn_1$ WF1 and in standard form. Straightforwardly, the result of applying the type 2 context configuration cn to cn_2 is the configuration $cn_2\ cn$. Define the configuration mapping $\mathtt{ten}$, which takes type 2 configurations to their type 1 counterparts by removing locations and queues from objects, and adding message containers, with the help of the inverse of the name representation map, $\mathtt{rep}^{-1}$. The result of applying the context cn to $\mathsf{bind}\ \overline{o}.cn_1$ can then be defined as the standard-form configuration $\mathsf{bind}\ \overline{o}'.\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$, where all free names originating from cn in $\mathsf{bind}\ \overline{o}.\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$ have been bound. Consequently, converse context closure requires that $cn_2\ cn\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{o}'.\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$ in this case.

Using the established equivalence, $\simeq$, we can now finally state the correctness property.

Theorem 2.7.4 (Correctness of the Type 2 Semantics)**.** *For all well-formed type 1 configurations* $\mathsf{bind}\ \overline{o}.cn$ *in standard form,* $\mathsf{bind}\ \overline{o}.cn \simeq \mathtt{net}(cn)$*.*

The proof of Theorem 2.7.4 proceeds by showing, with the help of Lemma 2.7.3, that the relation

$$\mathcal{R} = \{(\mathsf{bind}\ \overline{o}.cn, cn') \mid \mathtt{net}(cn) \equiv_2 cn'\},$$

where $\mathsf{bind}\ \overline{o}.cn$ is WF1 and in standard form, and cn' is WF2, is a conflated witness relation. This is sufficient, since the identity relation is included in $\equiv_2$.

2.8 Scheduling

Soundness and full abstraction is a useful validation that the network-aware semantics induces the same behavior on μABS programs as the reference semantics, but the properties rely on the inherent nondeterminism of the semantics. An implementation of the network-aware semantics must resolve the choices by introducing a scheduler which removes some transitions. Resolving the choices involves making crucial tradeoffs between management overhead and performance. For example, if all nodes exchange routing tables with their neighboring nodes with a high frequency, or quickly following location changes, routes can be assumed to always be close to optimal, but at the cost of a large messaging overhead. Similarly, if objects quickly change location when a node is overloaded, the variance in load between nodes can be kept low and the load evenly balanced, at the cost of, e.g., task execution throughput.

The difficulty with scheduling is preemptive choice. For instance, in the type 2 semantics, a scheduler may force two messages that in the nondeterministic semantics are causally independent to be received in some given order. This phenomenon makes contextual equivalence and bisimulation-oriented methods in general inapplicable. One might hope to be able to devise schedulers that correspond to each other at both the type 1 and type 2 levels. We argue, however, that this is undesirable: a scheduler at the type 1 level may require global coordination across the entire network to be enforced at the type 2 level, e.g., to speed up message transmission across some links and slow them down correspondingly across others—without these links having any network proximity constraints whatsoever. This is exactly the kind of global synchronization overhead the type 2 semantics is designed to avoid. We therefore deliberately restrict attention to scheduling at the type 2 level. However, even in the presence of a scheduler at this level, we can still draw strong conclusions on faithfulness to the reference semantics, as we demonstrate below using a contextual simulation preorder in place of contextual equivalence. The idea is to view schedulers abstractly as predicates on type 2 configuration transition histories.

Definition 2.8.1 (Execution, Scheduler)**.** An *execution*, of either the type 1 or type 2 semantics, is a sequence of well-formed configurations $\rho = cn_1 \cdots cn_n$, such that, for $i : 1 \leq i < n$, it holds that $cn_i \to cn_{i+1}$. Let $\langle cn \rangle$ be the singleton execution consisting of only the configuration *cn*. A *scheduler* is a predicate $\mathcal{S}$ on type 2 executions, such that

- $\mathcal{S}(\langle cn \rangle)$ for all $\langle cn \rangle$, i.e., a scheduler kicks in only once an execution is started, and
- if $\mathcal{S}(cn_1 \cdots cn_n)$ and there exists a cn_{n+1} such that $cn_n \to cn_{n+1}$, then we have $\mathcal{S}(cn_1 \cdots cn_n\, cn_{n+1})$ for precisely one such cn_{n+1}.

That is, a scheduler is a device that determinizes type 2 executions. We define transition systems on executions, such that $\rho \to \rho'$ whenever $\rho = cn_1 \cdots cn_n$ and $\rho' = cn_1 \cdots cn_n\, cn_{n+1}$. The observation predicate $\rho \downarrow obs$, and application of a context configuration, is defined for executions similarly. A *scheduled transition system* is a transition system on type 2 executions, where, if we have $\rho \to \rho'$, $\mathcal{S}(\rho)$ and $\mathcal{S}(\rho')$ holds.

Let $\mathcal{R}$ be a relation on type 2 executions, scheduled by the scheduler $\mathcal{S}$, and unscheduled type 1 executions. Suppose $\mathcal{R}$ satisfies reduction closure, context closure and barb preservation, but that $\mathcal{R}^{-1}$ does not necessarily satisfy the converse properties. Assume the ($\mathcal{S}$-scheduled) type 2 execution ρ and the (unscheduled) type 1 execution ρ' are related by such an $\mathcal{R}$. We then say that ρ and ρ' are in the contextual simulation preorder $\sqsubseteq$, written $\rho \sqsubseteq \rho'$, and we obtain from Theorem 2.7.4 the following corollary.

Corollary 2.8.2. *For all well-formed type 1 configurations* $\mathsf{bind}\ \overline{o}.cn$ *in standard form,* $\langle \mathtt{net}(cn) \rangle \ \sqsubseteq\ \langle \mathsf{bind}\ \overline{o}.cn \rangle$.

Intuitively, Corollary 2.8.2 says that a scheduled execution in the network-aware semantics always maps to some specific (valid) execution in the network-oblivious semantics.

2.9 Related Work

Much work has been done on object/component mobility in the π-calculus tradition, and on the implementation of high-level object or process-oriented languages in terms of more efficiently implementable low-level calculi. Sewell et al. [184], following earlier work on Pict [168], Fournet's distributed join-calculus [76], and JoCaml [42], implement and prove correct a compiler for Nomadic Pict, a prototype language similar to the μABS language: principally asynchronous message passing between named, location-oblivious processes. The correctness argument for Nomadic Pict with a central forwarding server scheme uses coupled simulation [162] to handle problems related to preemptive choice that arise due to the use of locks. In comparison, the use of location independent routing allows us to use contextual equivalence in place of coupled simulation, and consequently, we obtain a simpler correctness proof.

In the Klaim project [19], compilers are implemented and proven correct for several variants of the Klaim language, using the Linda tuple space communication model and a centralized name server to identify local tuple servers. The Oz kernel language [189] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [121] adds to this a notion of locality with separate failure and mobility semantics, but no real distribution or communication semantics is given; long distance communication is reduced to explicit manipulation of located agents, in the style of the Ambient Calculus [28].

Standard ABS [103] provides a model of concurrent objects, related to the cobox model [183] and Creol [104], but without any concept of locations or communication medium. Another difference is that the ABS unit of concurrency is an object group rather than a task, resulting in a more intuitive programming model without data races. A model with many concurrent tasks, as described here, is still feasible to use by programmers, and can allow more efficient execution on multi-core nodes [88]. Substantial work has been done in the HATS project on ABS and its extensions, e.g., towards software product lines [182]. Johnsen et al. [105] propose an extension of ABS with deployment components for explicit resource management. In contrast, the setting is not inherently decentralized as in our case, and the component model abstracts away from message distribution and routing. Viewing network nodes as deployment components, the model is amenable to the location independent routing approach for message-passing, which may be useful for lower-level software systems modeling.

We have extended our analysis of the rudimentary μABS language to a setting with more sophisticated constructs for communication, namely, futures as placeholders for method call return values [51]. This adds many complications related to global consistency of future values, but contextual equivalence can still be proved following the outline given here. Past correctness analyses for object languages with futures have been carried out, e.g., by Caromel et al. [33], who prove a confluence result for their language of asynchronous sequential processes, however without an explicit treatment of distribution, communication, and routing.

We have also enhanced a variant of the more extensive core ABS language, which includes a type system, to make it network aware, as described in Appendix B. We refer to the combination of the core ABS syntax and the corresponding network-aware semantics as ABS-NET, and have used it to investigate decentralized runtime adaptation to requirements on node load, link load, and message latency through decentralized object migration for some simple programs [159]. In that work, we implemented ABS-NET in a simulator in Java using TCP sockets. The ABS-NET semantics is split into a language interpreter layer and a language-independent node controller layer, not unlike the actors and meta-actors in the architecture of Mechitov et al. [139].

On the one hand, our approach reduces complexity by doing away with some conventional parts of the network stack, i.e., changing the boundaries between layers. This approach has found success in other domains, such as data storage systems, where it has been referred to as *telescoping* a stack of layers [24]. Besides simplicity, rethinking layers can allow for significant performance gains for applications through reduced overhead, as demonstrated by Marinos et al. [135] with a custom network stack tailored for use by web servers. On the other hand, throwing away OSI network layers 3 and above may be an excessive price to pay, and it may turn out to be infeasible to amend current IP schemes in the direction proposed here. However, the architecture of the future Internet is currently very much in flux. It is possible today to build large scale non-IP networks with only layer 2 connectivity, sufficient to bootstrap a location independent scheme such as ours. The simplicity of formal reasoning when using our approach in comparison to the task of formally verifying, e.g., IP and TCP [20], suggests that currently ongoing work on verification of low-level software, along the lines of seL4 [109], can be extended to include fully networked operating systems and hypervisors.

In the Cloud Computing paradigm, a pool of network-interconnected computing resources are shared between different applications. Most proposed solutions to resource control for such pools, called clouds, are centralized, and can thus be expected to scale only to systems with in the order of thousands of nodes [102]. Together with a decentralized scheduling strategy that provides resource control through object migration, similar to those that we have investigated for ABS-NET [159], our approach provides a building block for a scalable, formalized Platform as a Service (PaaS), that can be implemented correctly with few assumptions on network capabilities. The fully local nature of the reduction rules ensures that implementation code can correspond closely to the formalization, minimizing the risk of errors. An implementation of our semantics that realizes a PaaS cloud, with a deployment spanning several physical locations, can use the hybrid approach alluded to in Section 2.1. In a more general routing scheme, weights can be attached to links, instead of simply relying on the number of hops to approximate distance to an object. Physical links between nodes inside a datacenter, used for communication without TCP/IP, can then be assigned, e.g., a fixed low weight, while links between "gateway" nodes in different datacenters, established using TCP over the Internet, can have higher weights, depending on their conventional routing distance.

The network-aware semantics assumes a fixed, static network throughout execution. Real-world networks are dynamic in at least two different ways. First, nodes can crash, then possibly recover, or deviate arbitrarily from prescribed behavior. Second, nodes can be added and shut down in a controlled way. In related work, described in Chapter 5, we consider the latter kind of dynamicity. We use a protocol reminiscent of a two-phase commit [187] to ensure objects can always be safely migrated away, and messages routed away, from nodes shutting down. The protocol in effect rules out simultaneous shutdown of neighboring nodes that have outstanding object-related messages between them, whence program-related state cannot disappear. Extended using suitable local criteria for connectivity, the protocol can also ensure networks remain connected after shutdowns, which is necessary for progress in program execution. To preserve object behavior in the face of node crash failures, some form of replication must be used for objects, tasks, and messages; the state checkpointing approach of Field and Varela [69] is one possibility. Handling of crashes can be formalized using failure detectors [36, 150].

2.10 Applications in Service Oriented Computing

The Service Oriented Computing (SOC) paradigm is reciprocal to Cloud Computing; the former deals with "computing of services", while the latter provides "services of computing" [204]. In SOC-based systems, some basic services are depended on by higher-level services, and must therefore have high availability and responsiveness. If these basic services are implemented using active objects in a language similar to ABS, and deployed in a cloud using our approach, they can become mobile across datacenters, achieve high availability and responsiveness by adaptive resource allocation, and accommodate growing use by scaling to large networks.

SOC services can be connected by message exchanges, through orchestration at higher levels of abstraction. In our approach to resource allocation, described in Chapter 4, all decisions are taken locally at runtime, which enables allocations of service components based on message exchange patterns. For example, groups of services deployed separately, that nevertheless interact frequently, can after some time execute on nodes in close proximity, lowering overall response times.

Current SOC systems make extensive use of web services [161]. PaaS providers such as Google and Amazon offer cloud-based deployment of applications realizing web services, with automatic resource provisioning based on demand. Yet, scaling down in, e.g., Google's App Engine, removes server instances without taking application state into account, since a shared-nothing architecture is assumed [2]. In our approach, application state in a node selected for shutdown can automatically migrate elsewhere. This offers service developers more flexibility in web application architecture. Pruning among active objects themselves can be done in a safe way using distributed garbage collection [201].

2.11 Conclusion

We have presented a sound and fully abstract semantics for a rudimentary object language, in terms of a network-aware execution model. Thanks in part to a novel explicit mixing of messaging and routing, we are able to present the model at a level where it can in principle be implemented in a provably correct fashion directly on top of silicon, or integrated in a hypervisor such as Xen [16], assuming reliable link layer (OSI layer 2) functionality only. Essentially, our approach is language independent, and can be applied to languages such as Core Erlang with only minor local changes, e.g., adding a pattern matching construct for selectively receiving messages.

In future work, we want to consider richer network models with features such as power control—allowing executions with adaptive power consumption—and various forms of node failure. The model can then be used as a platform for language-based studies of load balancing and resource adaptation. While we have investigated strategies for decentralized adaptation to requirements such as node load and link load [159], it remains to address the full spectrum of implementation-level concerns such as crash failures, Byzantine failures, garbage collection, and buffer management. In addition, the routing scheme must be reconsidered. Distance vector routing suffers from fundamental scalability and security problems, and needs attention in light of recent progress on compact routing [186].

Acknowledgements

We thank the anonymous reviewers of the paper, and its previous version, for their comments and suggestions. We also thank our colleagues in the HATS project for useful discussions.

2.12 Proofs

Proposition 2.4.1. *Suppose $cn \to cn'$. Then, the following holds:*

1. $\mathtt{fn}(cn') \subseteq \mathtt{fn}(cn)$.

2. *If* $\mathsf{o}(o, a) \preceq cn$, *then* $\mathsf{o}(o, a') \preceq cn'$ *for some object environment* a'.

Proof. For the first property, note that no structural identity nor any reduction rule allows an OID to escape its binder. The result follows. For the second property, inspection reveals that no rule removes an object container from a configuration, or changes an OID. □

Proposition 2.4.4. *Let* cn *be a type 1 configuration. Then, the following holds:*

1. *If* cn *is a type 1 initial configuration, then* cn *is WF1.*

2. *If* cn *is WF1 and* $cn \to cn'$, *then* cn' *is WF1.*

Proof. By inspection of the definitions and rules. □

Proposition 2.5.3. *Suppose* $cn \to cn'$. *Then, the following holds:*

1. *If* $\mathsf{n}(u, t) \preceq cn$, *then* $\mathsf{n}(u, t') \preceq cn'$ *for some* t'.

2. *If* $\mathsf{l}(u, q, u') \preceq cn$, *then* $\mathsf{l}(u, q', u') \preceq cn'$ *for some* q'.

3. *If* $obj = \mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, *then there is an object container* $obj' = \mathsf{o}(o', a', u', q'_{in}, q'_{out}) \preceq cn'$ *(the* derivative *of* obj *in* cn'*) such that* $o' = o$ *and for all* x, *if* $a(x)\downarrow$, *then* $a'(x)\downarrow$.

Proof. By inspection of the definitions and rules. □

Proposition 2.5.6. *Let* cn *be a configuration. Then, the following holds:*

1. *If* cn *is a type 2 initial configuration, then* cn *is WF2.*

2. *If* cn *is WF2 and* $cn \to cn'$, *then* cn' *is WF2.*

Proof. Similar to the proof of Proposition 2.4.4. □

Proposition 2.6.3. *Algorithm 1 terminates.*

Proof. In each iteration of the outermost loop of Algorithm 1, exactly one message is enqueued on each proper link, and at least one message is dequeued from all link queues. MSG-RCV, MSG-DELAY-1, and OBJ-RCV cause messages to leave the link queues, except for external messages, which are moved to the self-loop queues. If the link queues have only routing table messages, the algorithm terminates in that iteration. If not, there must be object messages or routable call messages in some link queue. Since no new object messages are enqueued, there must some number of iterations n_0 after which all object messages have been received via OBJ-RCV and the associated object OIDs o registered on some node u so that $t(o) = (u, 0)$.

Let m_0 be the size of the largest link queue at the point which there are no object messages in transit. After $n_0 + m_0 + 1$ iterations, each node u has received at least one table update from each of its neighbors u', and the last table update applied to u has $t(o) = 0$. As a result, at point $n_0 + m_0 + 1$ each node u has $t(o) = (u', 1)$ whenever the host of o is u' and the minimal length path from u to u' has length 1. The entry of the routing table of u for o will not change from that point onwards. We say that those entries are *stable*. Proceeding, let m_1 be the length of the largest link queue at point $n_0 + m_0 + 1$. After $n_0 + m_0 + 1 + m_1 + 1$ iterations each routing table entry with length 2 (or less) will be stable. In the limit, each entry will be stable. It follows that Algorithm 1 must terminate, since, once routing has stabilized, rule MSG-ROUTE can only be applied a finite number of times before a routable message will be delivered. There is no chance of routable messages getting stuck in self-loop queues, since they are continuously shuffled using MSG-DELAY-3.

The only detail remaining to be checked is that a message can always be read from a link. Table and object messages can always be delivered, and call messages can also always be delivered, if nothing else to the self-loop link, in which case the routing table is not up-to-date or the message is external. This is the only case where MSG-DELAY-1 is used. This completes the argument. □

Proposition 2.6.5. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then:*

1. *$cn \to^* cn'$,*
2. *cn' is in stable form,*
3. *$\mathtt{graph}(cn') = \mathtt{graph}(cn)$,*
4. *$\mathsf{t}(cn') = \mathsf{t}(cn)$,*
5. *$\mathsf{o}(cn') \cong_1 \mathsf{o}_1(cn)$, and*
6. *$\mathsf{m}(cn') = \mathsf{m}_1(cn)$.*

Proof. Property 1 and 3 are immediate. Property 2 can be read out of the termination proof. For the remaining three properties, observe first that t, o_1, and m_1 are all invariant under the transitions used in Algorithm 1. The equations follow by noting that only external messages (and so no object closures) are in transit in cn'. □

Corollary 2.6.7. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then $cn \equiv_1 cn'$.*

Proof. We have $\mathcal{A}_1(cn) \rightsquigarrow cn' \; \mathcal{R} \; cn' \leftsquigarrow \mathcal{A}_1(cn')$. □

Lemma 2.6.8. *$\equiv_1$ is reduction closed.*

Proof. Suppose $cn_1 \equiv_1 cn_2$, where both cn_1 and cn_2 are WF2. Assume $cn_1 \to cn_1'$; we need to find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \equiv_1 cn_2'$. We proceed by case analysis on the transition $cn_1 \to cn_1'$, eliding uses of CTXT-1. For the cases T-SEND, T-RCV, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, OBJ-RCV, MSG-DELAY-3, and OBJ-REG, we take $cn_2' = cn_2$, since in those cases the stable form is unaffected, i.e., $cn_1 \equiv_1 cn_1'$.

The remaining cases include the rules for sequential control, MSG-SEND, CALL-SEND-2, MSG-DELAY-2, CALL-RCV-2, NEW-2, and OBJ-SEND. The rules for sequential control are handled in a structurally similar way; take WFIELD as an example, with a transition of the form

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$$

where $[\![e]\!]_{(a,l)} = v$ and $x \in \mathtt{dom}(a)$. Consider cn_2'' such that $\mathcal{A}_1(cn_2) \leadsto cn_2''$. By the definition of $\equiv_1$, there is a task container $\mathsf{t}(o, l, s)$ and an object container $\mathsf{o}(o, a, u, q_{in}', q_{out})$ in cn_2''. Hence, it is possible to perform a transition

$$cn'\ \mathsf{o}(o, a, u, q_{in}', q_{out})\ \mathsf{t}(o, l, x = e; s) \to cn'\ \mathsf{o}(o, a[v/x], u, q_{in}', q_{out})\ \mathsf{t}(o, l, s)$$

and we have

$$cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s) \equiv_1 cn'\ \mathsf{o}(o, a[v/x], u, q_{in}', q_{out})\ \mathsf{t}(o, l, s),$$

as needed, setting cn_2' to the right-hand side.

MSG-SEND: Consider a transition of the form

$$\begin{aligned} & cn\ \mathsf{n}(u, t)\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{l}(u, q, u') \\ \to\quad & cn\ \mathsf{n}(u, t)\ \mathsf{o}(o, a, u, q_{in}, \mathsf{deq}(q_{out}))\ \mathsf{l}(u, \mathsf{enq}(msg, q), u') \end{aligned}$$

where $\mathtt{hd}(q_{out}) = msg$, $\mathtt{dst}(msg) = o'$, and $\mathtt{nxt}(o', t) = u'$. Consider cn_2'' such that $\mathcal{A}_1(cn_2) \leadsto cn_2''$. By the definition of $\equiv_1$, there are containers $\mathsf{o}(o, a, u, q_{in}', q_{out})$, $\mathsf{n}(u, t')$, and $\mathsf{l}(u, q', u'')$, such that $\mathtt{nxt}(o', t') = u''$. Hence, it is possible to perform a transition

$$\begin{aligned} & cn'\ \mathsf{n}(u, t')\ \mathsf{o}(o, a, u, q_{in}', q_{out})\ \mathsf{l}(u, q', u'') \\ \to\quad & cn'\ \mathsf{n}(u, t')\ \mathsf{o}(o, a, u, q_{in}', \mathsf{deq}(q_{out}))\ \mathsf{l}(u, \mathsf{enq}(msg, q'), u'') \end{aligned}$$

and we have

$$\begin{aligned} & cn\ \mathsf{n}(u, t)\ \mathsf{o}(o, a, u, q_{in}, \mathsf{deq}(q_{out}))\ \mathsf{l}(u, \mathsf{enq}(msg, q), u') \\ \equiv_1\quad & cn'\ \mathsf{n}(u, t')\ \mathsf{o}(o, a, u, q_{in}', \mathsf{deq}(q_{out}))\ \mathsf{l}(u, \mathsf{enq}(msg, q'), u''), \end{aligned}$$

as needed, setting cn_2' to the right-hand side.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned} & cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, e_1!m(\overline{e_2}); s) \\ \to\quad & cn\ \mathsf{o}(o, a, u, q_{in}, \mathsf{enq}(\mathsf{call}(o', m, \overline{v}), q_{out}))\ \mathsf{t}(o, l, s) \end{aligned}$$

where $[\![e_1]\!]_{(a,l)} = o'$ and $[\![\overline{e_2}]\!]_{(a,l)} = \overline{v}$. Consider cn_2'' such that $\mathcal{A}_1(cn_2) \leadsto cn_2''$. By the definition of $\equiv_1$, there is a task container $\mathsf{t}(o, l, e_1!m(\overline{e_2}); s)$ and an object container $\mathsf{o}(o, a, u', q_{in}', q_{out})$ in cn_2''. Hence, it is possible to perform a transition

$$\begin{aligned} & cn'\ \mathsf{o}(o, a, u', q_{in}', q_{out})\ \mathsf{t}(o, l, e_1!m(\overline{e_2}); s) \\ \to\quad & cn'\ \mathsf{o}(o, a, u', q_{in}', \mathsf{enq}(\mathsf{call}(o', m, \overline{v}), q_{out}))\ \mathsf{t}(o, l, s) \end{aligned}$$

and we have

$$\begin{aligned} & cn\ \mathsf{o}(o, a, u, q_{in}, \mathsf{enq}(\mathsf{call}(o', m, \overline{v}), q_{out}))\ \mathsf{t}(o, l, s) \\ \equiv_1 \quad & cn'\ \mathsf{o}(o, a, u', q'_{in}, \mathsf{enq}(\mathsf{call}(o', m, \overline{v}), q_{out}))\ \mathsf{t}(o, l, s), \end{aligned}$$

as needed, setting cn'_2 to the right-hand side.

The other cases are proved similarly. □

Lemma 2.6.9. $\equiv_1$ *is context closed.*

Proof. Suppose that $cn_1 \equiv_1 cn_2$. Then, cn_1 and cn_2 are WF2. Assume $cn_1\ cn$ is WF2 for a context configuration cn. We first show that $cn_2\ cn$ is WF2. For OID Uniqueness, it suffices to consider the case where we have

$$\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn_2\ ,\ \mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq cn.$$

Then, if $o_1 = o_2$, there is a clash between OIDs in cn_1 and cn, since there is an object container with OID o_1 in cn_1 by property 5 of Proposition 2.6.5. For Task-Object existence, it suffices to consider the case where $\mathsf{t}(o, l, s) \preceq cn$; then, if there is no object container with OID o in $cn_2\ cn$, there is no such container in cn_1 either, violating WF2. For Object-Node Existence, suppose $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, but that there is no vertex u in $\mathtt{graph}(cn_2\ cn)$; since the graphs for cn_1 and cn_2 coincide, and there are no nodes in cn, this means that there is no such node in $\mathtt{graph}(cn_1\ cn)$, violating the WF2 assumption. For Buffer Cleanliness, it suffices to note that the property distributes over configuration composition. For Local Routing Consistency, note again that cn introduces no nodes or links, and cn_2 is WF2. For External OID, note that cn cannot have an object container with OID *ext* since $cn_1\ cn$ is WF2, and that cn does not change the nodes or links.

It remains to show that we have $cn_1\ cn\ \equiv_1\ cn_2\ cn$. The WF2 property is immediate. Suppose $\mathcal{A}_1(cn_1\ cn) \leadsto cn'_1$ and $\mathcal{A}_1(cn_2\ cn) \leadsto cn'_2$. It suffices to show $cn'_1\ \mathcal{R}_1\ cn'_2$. We prove the conditions in turn.

$$\begin{aligned} \mathtt{graph}(cn'_1) = \mathtt{graph}(cn_1\ cn) = \mathtt{graph}(cn_1) = \\ \mathtt{graph}(cn_2) = \mathtt{graph}(cn_2\ cn) = \mathtt{graph}(cn'_2). \end{aligned}$$

$$\begin{aligned} \mathsf{t}(cn'_1) = \mathsf{t}(cn_1\ cn) = \mathsf{t}(cn_1) \cup \mathsf{t}(cn) = \\ \mathsf{t}(cn_2) \cup \mathsf{t}(cn) = \mathsf{t}(cn_2\ cn) = \mathsf{t}(cn'_2). \end{aligned}$$

$$\begin{aligned} \mathsf{o}(cn'_1) \cong_1 \mathsf{o}_1(cn_1\ cn) = \mathsf{o}_1(cn_1) \cup \mathsf{o}_1(cn) = \\ \mathsf{o}_1(cn_2) \cup \mathsf{o}_1(cn) = \mathsf{o}_1(cn_1\ cn) \cong_1 \mathsf{o}(cn'_2). \end{aligned}$$

$$\begin{aligned} \mathsf{m}(cn'_1) = \mathsf{m}_1(cn_1\ cn) = \mathsf{m}_1(cn_1) \cup \mathsf{m}_1(cn) = \\ \mathsf{m}_1(cn_2) \cup \mathsf{m}_1(cn) = \mathsf{m}_1(cn_2\ cn) = \mathsf{m}(cn'_2). \end{aligned}$$

The proof of converse context closure is symmetric. □

Proposition 2.6.10. $\equiv_1$ *is a type 2 witness relation.*

Proof. We have that $\equiv_1$ is reduction closed by Lemma 2.6.8 and context closed by Lemma 2.6.9. Hence, it suffices to show barb preservation in both directions. Suppose $cn_1 \equiv_1 cn_2$ and $cn_1 \downarrow obs$, where $obs = ext!m(\overline{v})$. Then, there is a message $\mathsf{call}(ext, m, \overline{v})$ at the head of some self-loop queue in cn_1. Consequently, after running Algorithm 1 on cn_2, this message will be found in some self-loop queue, from which it can be brought to the head by means of repeated application of MSG-DELAY-3. Thus, $cn_2 \Downarrow obs$. The proof of converse barb preservation is symmetric. ☐

Corollary 2.6.11. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then $cn \simeq_2 cn'$.*

Proof. By Proposition 2.6.10 and Corollary 2.6.7. ☐

Proposition 2.6.12. *Algorithm 2 terminates.*

Proof. Routing is stable after each run of Algorithm 1, and none of the rules applied in the first inner loop affect routing stability. Also, after the first run of Algorithm 1, links contain only external calls. Whenever an object out-queue is nonempty, one of MSG-SEND or MSG-DELAY-2 will be enabled. By Buffer Cleanliness, CALL-RCV-2 will be applicable if the object in-queue is nonempty, decreasing in-queue size by one. Thus, when the first while loop is exited, object queues are empty. The second while loop terminates when all objects not yet at u have been put on the wire. At the end of each outer loop, routing is stabilized and link queues emptied (except for external messages). Once emptied, out-queues remain empty. In-queues may contain messages at the start of the second iteration, but after that, only external messages remain in either link or object queues, except for object closures, which are consumed once they reach u. ☐

Proposition 2.6.14. *If cn is WF2 and $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then:*

1. $cn \rightarrow^* cn'$,

2. *cn' is in normal form,*

3. $\mathtt{graph}(cn') = \mathtt{graph}(cn)$,

4. $\mathsf{t}(cn') = \mathsf{t}_2(cn)$,

5. $\mathsf{o}(cn') = \mathsf{o}_2(cn)$, *and*

6. $\mathsf{m}(cn') = \mathsf{m}_2(cn)$.

Proof. Property 1 and 3 are immediate. We consider the requirements of property 2 in turn. By virtue of Proposition 2.6.5, cn' has stable routing and external link messages. By the termination requirements of Algorithm 2, all object queues are empty, meaning that all call messages have been delivered, and there are no object containers in transit, yielding $\mathsf{t}(cn') = \mathsf{t}_2(cn')$, $\mathsf{o}(cn') = \mathsf{o}_2(cn')$, and $\mathsf{m}(cn') = \mathsf{m}_2(cn')$.

For property 4, observe first that the function t_2 is invariant under transitions used in Algorithm 2. On termination of Algorithm 2, only external messages are in transit, and since no rule causes a task to be modified, the property follows.

For property 5, suppose $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}(cn')$. We need to show that $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}_2(cn)$. By definition, $q_{in,2} = q_{out,2} = \varepsilon$ and $u_2 = u_0$. We know that there is an object container $\mathsf{o}(o, a', u', q_{in}, q_{out}) \preceq cn$, as there is a one-to-one correspondence between object containers in pre- and poststate for each transition used in Algorithm 2. We also know that $a'(x) = a_2(x)$ for all x, which suffices.

For property 6, note again that all object queues are empty, link queues only have external messages, and finally that no new external messages are generated by the rules used in Algorithm 2. □

Corollary 2.6.16. $\equiv_1 \subseteq \equiv_2$.

Proof. If $cn_1 \equiv_1 cn_2$, the two configurations have the same task containers, and the same object bound messages. In addition, there is a one-to-one mapping between object containers where identifiers and object environments coincide. The result follows by noting that any remaining differences between the containers will disappear after running Algorithm 2. □

Corollary 2.6.17. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \equiv_2 cn'$.*

Proof. By Proposition 2.6.14. □

Lemma 2.6.18. $\equiv_2$ *is reduction closed.*

Proof. Suppose $cn_1 \equiv_2 cn_2$, where cn_1 and cn_2 are WF2. Assume $cn_1 \to cn_1'$; we need to find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \equiv_2 cn_2'$. We proceed by case analysis on the transition $cn_1 \to cn_1'$, eliding uses of CTXT-1. For the cases T-SEND, T-RCV, MSG-SEND, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, MSG-DELAY-2, MSG-DELAY-3, CALL-RCV-2, OBJ-REG, OBJ-SEND, and OBJ-RCV, we take $cn_2' = cn_2$, since in those cases the normal form is unaffected, i.e., $cn_1 \equiv_2 cn_1'$. The remaining cases include the rules for sequential control, CALL-SEND-2, and NEW-2. The rules for sequential control are handled in a structurally similar way; take WFIELD as an example, with a transition of the form

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s),$$

where $[\![e]\!]_{(a,l)} = v$ and $x \in \mathtt{dom}(a)$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By the definition of $\equiv_2$, there is a task container $\mathsf{t}(o, l, x = e; s)$ and an object container $\mathsf{o}(o, a, u', \varepsilon, \varepsilon)$ in cn_2''. Hence, it is possible to perform a transition

$$cn' \ \mathsf{o}(o, a, u', \varepsilon, \varepsilon) \ \mathsf{t}(o, l, x = e; s) \to cn' \ \mathsf{o}(o, a[v/x], u', \varepsilon, \varepsilon) \ \mathsf{t}(o, l, s),$$

and we have that

$$cn \ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out}) \ \mathsf{t}(o, l, s) \equiv_2 cn' \ \mathsf{o}(o, a[v/x], u', \varepsilon, \varepsilon) \ \mathsf{t}(o, l, s),$$

as needed, setting cn_2' to the right-hand side.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned} & cn \ \mathsf{o}(o, a, u, q_{in}, q_{out}) \ \mathsf{t}(o, l, e_1!m(\overline{e_2}); s) \\ \to \quad & cn \ \mathsf{o}(o, a, u, q_{in}, \mathsf{enq}(\mathsf{call}(o', m, \overline{v}), q_{out})) \ \mathsf{t}(o, l, s) \end{aligned}$$

where $[\![e_1]\!]_{(a,l)} = o'$ and $[\![\overline{e_2}]\!]_{(a,l)} = \overline{v}$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By the definition of $\equiv_2$, there is a task container $\mathsf{t}(o, l, e_1!m(\overline{e_2}); s)$ and an object container $\mathsf{o}(o, a, u', \varepsilon, \varepsilon)$ in cn_2''. Hence, it is possible to perform a transition

$$\begin{aligned} & cn' \ \mathsf{o}(o, a, u', \varepsilon, \varepsilon) \ \mathsf{t}(o, l, e_1!m(\overline{e_2}); s) \\ \to \quad & cn' \ \mathsf{o}(o, a, u', \varepsilon, \mathsf{enq}(\mathsf{call}(o', m, \overline{v}), \varepsilon)) \ \mathsf{t}(o, l, s) \end{aligned}$$

and we have

$$\begin{aligned} & cn \ \mathsf{o}(o, a, u, q_{in}, \mathsf{enq}(\mathsf{call}(o', m, \overline{v}), q_{out})) \ \mathsf{t}(o, l, s) \\ \equiv_2 \quad & cn' \ \mathsf{o}(o, a, u', \varepsilon, \mathsf{enq}(\mathsf{call}(o', m, \overline{v}), \varepsilon)) \ \mathsf{t}(o, l, s), \end{aligned}$$

as needed, setting cn_2' to the right-hand side.

NEW-2: Consider a transition of the form

$$\begin{aligned} & cn \ \mathsf{o}(o, a, u, q_{in}, q_{out}) \ \mathsf{t}(o, l, \mathbf{new} \ C(\overline{e}); s) \\ \to \quad & cn \ \mathsf{o}(o, a, u, q_{in}, q_{out}) \ \mathsf{t}(o, l[o'/x], s) \ \mathsf{o}(o', a', u, \varepsilon, \varepsilon) \end{aligned}$$

where $\mathtt{newo}(u) = o'$, $[\![\overline{e}]\!]_{(a,l)} = \overline{v}$, and $\mathtt{init}(C, \overline{v}, o') = a'$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By the definition of $\equiv_2$, there is a task container $\mathsf{t}(o, l, \mathbf{new} \ C(\overline{e}); s)$ and an object container $\mathsf{o}(o, a, u', \varepsilon, \varepsilon)$ in cn_2''. Hence, it is possible to perform a transition

$$\begin{aligned} & cn' \ \mathsf{o}(o, a, u', \varepsilon, \varepsilon) \ \mathsf{t}(o, l, \mathbf{new} \ C(\overline{e}); s) \\ \to \quad & cn' \ \mathsf{o}(o, a, u', \varepsilon, \varepsilon) \ \mathsf{t}(o, l[o'/x], s) \ \mathsf{o}(o', a', u', \varepsilon, \varepsilon) \end{aligned}$$

and we have

$$\begin{aligned} & cn \ \mathsf{o}(o, a, u, q_{in}, q_{out}) \ \mathsf{t}(o, l[o'/x], s) \ \mathsf{o}(o', a', u, \varepsilon, \varepsilon) \\ \equiv_2 \quad & cn' \ \mathsf{o}(o, a, u', \varepsilon, \varepsilon) \ \mathsf{t}(o, l[o'/x], s) \ \mathsf{o}(o', a', u', \varepsilon, \varepsilon), \end{aligned}$$

as needed, setting cn_2' to the right-hand side. □

Lemma 2.6.19. $\equiv_2$ *is context closed.*

Proof. Suppose $cn_1 \equiv_2 cn_2$. Then, cn_1 and cn_2 are WF2. Assume $cn_1\ cn$ is WF2 for a context configuration cn. We first show that $cn_2\ cn$ is WF2. For OID Uniqueness, it suffices to consider the case where

$$\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn_2, \mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq cn.$$

Then, if $o_1 = o_2$, there is a clash between OIDs in cn_1 and cn, since there is an object container with OID o_1 in cn_1 by normal form equivalence, but this is ruled out by $cn_1\ cn$ being WF2. For Task-Object existence, it suffices to consider the case where $\mathsf{t}(o, l, s) \preceq cn$; then, if there is no object container with OID o in $cn_2\ cn$, there is no such container in cn_1 either, violating WF2. For Object-Node Existence, suppose we have $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, but that there is no vertex u in $\mathtt{graph}(cn_2\ cn)$; since the graphs for cn_1 and cn_2 coincide, and there are no nodes in cn, this means that there is no such node in $\mathtt{graph}(cn_1\ cn)$, violating the WF2 assumption. For Buffer Cleanliness, it suffices to note that the property distributes over configuration composition. For Local Routing Consistency, note again that cn introduces no nodes or links, and cn_2 is WF2. For External OID, note that cn cannot have an object container with OID *ext* since $cn_1\ cn$ is WF2, and that cn does not change the nodes or links.

It remains to show that $cn_1\ cn \equiv_2 cn_2\ cn$. The WF2 property is immediate. Suppose $\mathcal{A}_2(cn_1\ cn) \rightsquigarrow cn_1'$ and $\mathcal{A}_2(cn_2\ cn) \rightsquigarrow cn_2'$. It suffices to show $cn_1'\ \mathcal{R}_2\ cn_2'$. We prove the conditions in turn. Note that since $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$, we have $\mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn)$.

$$\begin{aligned}\mathtt{graph}(cn_1') = \mathtt{graph}(cn_1\ cn) = \mathtt{graph}(cn_1) = \\ \mathtt{graph}(cn_2) = \mathtt{graph}(cn_2\ cn) = \mathtt{graph}(cn_2').\end{aligned}$$

$$\mathsf{t}(cn_1') = \mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn) = \mathsf{t}(cn_2').$$

$$\begin{aligned}\mathsf{o}(cn_1') = \mathsf{o}_2(cn_1\ cn) = \mathsf{o}_2(cn_1) \cup \mathsf{o}_2(cn) = \\ \mathsf{o}_2(cn_2) \cup \mathsf{o}_2(cn) = \mathsf{o}_2(cn_2\ cn) = \mathsf{o}(cn_2').\end{aligned}$$

$$\begin{aligned}\mathsf{m}(cn_1') = \mathsf{m}_2(cn_1\ cn) = \mathsf{m}_2(cn_1) \cup \mathsf{m}_2(cn) = \\ \mathsf{m}_2(cn_2) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2\ cn) = \mathsf{m}(cn_2').\end{aligned}$$

The proof of converse context closure is symmetric. □

Proposition 2.6.20. $\equiv_2$ *is a type 2 witness relation.*

Proof. We have that $\equiv_2$ is reduction closed by Lemma 2.6.18 and context closed by Lemma 2.6.19. Hence, it suffices to show barb preservation in both directions. Suppose $cn_1 \equiv_2 cn_2$ and $cn_1 \downarrow obs$. Then, there is a message with destination *ext* at the head of some self-loop queue in cn_1. Consequently, after running Algorithm 2 on cn_2, this message will be found in some self-loop queue, from where it can be brought to the head by means of repeated application of MSG-DELAY-3. Thus, $cn_2 \Downarrow obs$. The proof of converse barb preservation is symmetric. □

Corollary 2.6.21. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \simeq_2 cn'$.*

Proof. None of the rules used in Algorithm 2 affects the shape of the normal form. Thus, if $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \equiv_2 cn'$. But then, $cn \simeq cn'$, by Proposition 2.6.20. □

Proposition 2.7.2. *If $\mathsf{bind}\ \overline{o}.cn$ is a WF1 configuration in standard form, then $\mathtt{net}(cn)$ is WF2.*

Proof. We consider the WF2 conditions in turn. OID Uniqueness and Task-Object Existence follows from the respective WF1 conditions and from how the name representation map is defined. Object-Node Existence holds since all objects are placed on the node u_0, which exists by the definition of cn_{graph}. Buffer Cleanliness holds since all object containers in $\mathtt{net}(cn)$ have empty queues. Local Routing Consistency follows from how routing tables in cn_{graph} are defined. External OID follows from the corresponding WF1 condition and from how routing tables are defined. □

Lemma 2.7.3. *Let $\mathsf{bind}\ \overline{o}.cn$ be a type 1 well-formed configuration in standard form. Then:*

1. *If $\mathsf{bind}\ \overline{o}.cn \rightarrow \mathsf{bind}\ \overline{o}'.cn'$, then for some cn'', $\mathtt{net}(cn) \rightarrow^* cn''$ and $cn'' \equiv_2 \mathtt{net}(cn')$.*
2. *If $\mathtt{net}(cn) \rightarrow cn''$, then for some $\overline{o}'$ and cn', $\mathsf{bind}\ \overline{o}.cn \rightarrow^* \mathsf{bind}\ \overline{o}'.cn'$ and $cn'' \equiv_2 \mathtt{net}(cn')$.*

Proof. 1. The proof is by case analysis on the possible transitions such that $\mathsf{bind}\ \overline{o}.cn \rightarrow \mathsf{bind}\ \overline{o}'.cn'$, eliding uses of CTXT-1 and CTXT-2. The remaining sequential control rules in Figure 2.1 are straightforward; consider, for instance, the rule WFIELD, which yields a transition of the form

$$\mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o, a)\ \mathsf{t}(o, l, x = e; s) \rightarrow \mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o, a[v/x])\ \mathsf{t}(o, l, s),$$

where $x \in \mathtt{dom}(a)$ and $v = [\![e]\!]_{(a,l)}$. We calculate:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=e;s))} \\
= & (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ \mathtt{net}(\mathsf{t}(o,l,x=e;s),\mathtt{rep}))(cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})(\mathtt{net}(\mathsf{t}(o,l,x=e;s),\mathtt{rep})(cn_{graph}))) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})(\mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=e;s)\ cn_{graph})) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=e;s)\ cn_{graph}) \\
\to & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a)[\mathtt{rep}(v)/x],u_0,\varepsilon,\varepsilon) \\
 & \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a[v/x]),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a[v/x])\ \mathsf{t}(o,l,s))
\end{array}$$

using the rule WFIELD-2 to derive the transition.

We proceed to the rules concerning messages and object creation.

CALL-SEND: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,e_1!m(\overline{e_2});s) \to \mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s)\ \mathsf{c}(o',m,\overline{v}),$$

where $o' = [\![e_1]\!]_{(a,l)}$ and $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$. We calculate:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,e_1!m(\overline{e_2});s))} \\
= & (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ \mathtt{net}(\mathsf{t}(o,l,e_1!m(\overline{e_2});s),\mathtt{rep}))(cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})(\mathtt{net}(\mathsf{t}(o,l,e_1!m(\overline{e_2});s),\mathtt{rep})(cn_{graph}))) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})(\mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),e_1!m(\overline{e_2});s)\ cn_{graph})) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),e_1!m(\overline{e_2});s)\ cn_{graph}) \\
\to & \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\mathsf{enq}(\mathsf{call}(\mathtt{rep}(o'),m,\mathtt{rep}(\overline{v})),\varepsilon)) \\
 & \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
\equiv_2 & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s) \\
 & \mathsf{send}(\mathsf{call}(\mathtt{rep}(o'),m,\mathtt{rep}(\overline{v})),cn_{graph})) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s)\ \mathsf{c}(o',m,\overline{v}))
\end{array}$$

using CALL-SEND-2.

CALL-RCV: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{c}(o,m,\overline{v}) \to \mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s),$$

where $l = \mathtt{locals}(o,m,\overline{v})$ and $s = \mathsf{body}(o,m)$. We calculate:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{c}(o,m,\overline{v}))} \\
= & (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ \mathtt{net}(\mathsf{c}(o,m,\overline{v}),\mathtt{rep}))(cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{send}(\mathsf{call}(\mathtt{rep}(o),m,\mathtt{rep}(\overline{v})),cn_{graph})) \\
\equiv_2 & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s))
\end{array}$$

NEW: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=\textbf{new}\ C(\overline{e});s) \rightarrow \mathsf{bind}\ o'\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a')$$

where $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$ and $a' = \mathtt{init}(C,\overline{v},o')$. We calculate:

$$\begin{aligned}
&\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=\textbf{new}\ C(\overline{e});s))\\
&\quad = \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\\
&\qquad\qquad \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=\textbf{new}\ C(\overline{e});s)\ cn_{graph})\\
&\quad \rightarrow \quad \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[o''/x],s)\\
&\qquad\qquad \mathsf{o}(o'',\mathtt{rep}'(a'),u_0,\varepsilon,\varepsilon)\ cn_{graph})\\
&\quad = \quad \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l[o''/x]),s)\\
&\qquad\qquad \mathsf{o}(o'',\mathtt{rep}'(a'),u_0,\varepsilon,\varepsilon)\ cn_{graph})\\
&\quad = \quad \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a'))
\end{aligned}$$

where $\mathtt{rep}' = \mathtt{rep}[o''/o']$ and $o'' = \mathtt{newo}(u_0)$. This completes the proof of property 1.

2. We proceed by cases on the type 2 rule applied to derive $\mathtt{net}(cn) \rightarrow cn''$, eliding uses of CTXT-1. The rules for sequential control are immediate, since it is straightforward to find cn' such that $\mathtt{net}(cn') = cn''$. For T-SEND, T-RCV, MSG-SEND, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, MSG-DELAY-2, MSG-DELAY-3, CALL-RCV-2, OBJ-REG, OBJ-SEND, and OBJ-RCV, we can set $\overline{o}' = \overline{o}$ and $cn' = cn$, since they have no effect on the normal form. We handle the two remaining cases as per below.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned}
&\mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),e_1!m(\overline{e_2});s)\ cn_{graph})\\
&\quad \rightarrow \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\mathsf{enq}(\mathsf{call}(\mathtt{rep}(o'),m,\mathtt{rep}(\overline{v}),\varepsilon)))\\
&\qquad\qquad \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph})
\end{aligned}$$

where $o' = [\![e_1]\!]_{(a,l)}$ and $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$. We have:

$$\mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,e_1!m(\overline{e_2});s) \rightarrow \mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s)\ \mathsf{c}(o',m,\overline{v})$$

and we calculate:

$$\begin{aligned}
&\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s)\ \mathsf{c}(o',m,\overline{v})\\
&\quad = \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\\
&\qquad\qquad \mathsf{send}(\mathsf{call}(\mathtt{rep}(o'),m,\mathtt{rep}(\overline{v})),cn_{graph}))\\
&\quad \equiv_2 \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\mathsf{enq}(\mathsf{call}(\mathtt{rep}(o'),m,\mathtt{rep}(\overline{v}),\varepsilon)))\\
&\qquad\qquad \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph})
\end{aligned}$$

NEW-2: Consider a transition of the form

$$\begin{aligned}
&\mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=\textbf{new}\ C(\overline{e});s)\\
&\quad \rightarrow \quad \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[o''/x],s)\\
&\qquad\qquad \mathsf{o}(o'',\mathtt{rep}'(a'),u_0,\varepsilon,\varepsilon)\ cn_{graph})
\end{aligned}$$

where we have $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \mathtt{init}(C, \overline{v}, o')$, $\mathtt{rep}' = \mathtt{rep}[o''/o']$, and $o'' = \mathtt{newo}(u_0)$. Then:

$$\begin{aligned} &\mathsf{bind}\ \overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = \mathbf{new}\ C(\overline{e}); s) \\ &\quad \rightarrow \quad \mathsf{bind}\ o'\,\overline{o}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a') \end{aligned}$$

and we calculate:

$$\begin{aligned} &\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a')) \\ &\quad = \quad \mathtt{net}(cn, \mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o), \mathtt{rep}'(a), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(\mathtt{rep}'(o), \mathtt{rep}'(l)[o''/x], s) \\ &\qquad\quad \mathsf{o}(o'', \mathtt{rep}'(a'), u_0, \varepsilon, \varepsilon)\ cn_{graph}) \end{aligned}$$

This completes the proof of property 2. □

Theorem 2.7.4. *For all well-formed type 1 configurations* $\mathsf{bind}\ \overline{o}.cn$ *in standard form,* $\mathsf{bind}\ \overline{o}.cn \simeq \mathtt{net}(cn)$.

Proof. We exhibit a conflated witness relation $\mathcal{R}$, defined as

$$\mathcal{R} = \{(\mathsf{bind}\ \overline{o}.cn, cn') \mid \mathtt{net}(cn) \equiv_2 cn'\},$$

where $\mathsf{bind}\ \overline{o}.cn$ is a WF1 configuration in standard form, and cn' is a WF2 configuration. Note that $(\mathsf{bind}\ \overline{o}.cn, \mathtt{net}(cn)) \in \mathcal{R}$, since the identity relation is included in $\equiv_2$. We show that $\mathcal{R}$ is a conflated type 1 and type 2 witness relation.

Suppose $\mathsf{bind}\ \overline{o}.cn_1\ \mathcal{R}\ cn_2$ (or the converse for $\mathcal{R}^{-1}$); then $\mathsf{bind}\ \overline{o}.cn_1$ is WF1 and in standard form, cn_2 is WF2, and $\mathtt{net}(cn_1) \equiv_2 cn_2$.

For reduction closure, assume $\mathsf{bind}\ \overline{o}.cn_1 \rightarrow \mathsf{bind}\ \overline{o}'.cn_1'$, where $\mathsf{bind}\ \overline{o}'.cn_1'$ is in standard form. Then, by property 1 of Lemma 2.7.3, $\mathtt{net}(cn_1) \rightarrow^* cn_1'' \equiv_2 \mathtt{net}(cn_1')$. This means that, for some cn_2', $cn_2 \rightarrow^* cn_2' \equiv_2 cn_1''$. Hence, by the transitivity of $\equiv_2$, $\mathsf{bind}\ \overline{o}.cn_1'\ \mathcal{R}\ cn_2'$. For converse reduction closure, assume $cn_2 \rightarrow cn_2'$. Then, $\mathtt{net}(cn_1) \rightarrow^* cn_2''$ and $cn_2' \equiv_2 cn_2''$. By property 2 of Lemma 2.7.3, this means that $\mathsf{bind}\ \overline{o}.cn_1 \rightarrow \mathsf{bind}\ \overline{o}'.cn_1'$ and $cn_2'' \equiv_2 \mathtt{net}(cn_1')$. Hence, by the transitivity of $\equiv_2$, $cn_2'\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{o}'.cn_1'$.

For barb preservation, assume $\mathsf{bind}\ \overline{o}.cn_1 \downarrow obs$. Then, $\mathtt{net}(cn_1) \Downarrow obs$, which by normal form equivalence implies that $cn_2 \Downarrow obs$, as needed. For converse barb preservation, assume $cn_2 \downarrow obs$. Then, by normal form equivalence, $\mathtt{net}(cn_1) \Downarrow obs$, and consequently $cn_1 \downarrow obs$, whereby $cn_1 \Downarrow obs$.

For context closure, assume $\mathsf{bind}\ \overline{o}'.cn_1\ cn$ is in standard form and WF1, and consider the configuration $\mathtt{net}(cn, \mathtt{rep})(cn_2)$, which in effect applies cn to cn_2. We first need to show that this resulting configuration is WF2. Object-Node Existence holds, since by the definition of $\mathtt{net}$, all objects in cn become attached to a node in cn_2. Buffer Cleanliness also holds by the definition of $\mathtt{net}$. For OID Uniqueness, it suffices to consider the case where $\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn_2$ and $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathtt{net}(cn, \mathtt{rep})(cn_2)$ but $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \not\preceq cn_2$; then, if $o_1 = o_2$, there is a corresponding clash in $\mathsf{bind}\ \overline{o}'.cn_1\ cn$, violating WF1. For

Task-Object Existence, assume $\mathtt{t}(o,l,s) \preceq \mathtt{net}(cn,\mathtt{rep})(cn_2)$ and $\mathtt{t}(o,l,s) \not\preceq cn_2$; then, if there is no object container $\mathtt{o}(o,a,u,q_{in},q_{out}) \preceq \mathtt{net}(cn,\mathtt{rep})(cn_2)$, there is no corresponding object container in $\mathsf{bind}\ \overline{o}'.cn_1\ cn$, violating WF1 for the tasks corresponding to $\mathtt{t}(o,l,s)$ in $\mathsf{bind}\ \overline{o}'.cn_1\ cn$. Local Routing Consistency holds since the composition does not add nodes or links and does not change routing tables. The first condition of External OID holds since *ext* is not allowed to be bound or defined in *cn*, and the second condition again holds since the context does not add network components.

It remains to show that $\mathsf{bind}\ \overline{o}'.cn_1\ cn\ \mathcal{R}\ \mathtt{net}(cn,\mathtt{rep})(cn_2)$. The network graphs of $\mathtt{net}(cn_1\ cn)$ and $\mathtt{net}(cn,\mathtt{rep})(cn_2)$ coincide, since applying *cn* does not introduce any nodes or links. Clearly, if and only if a task or non-external message is in *cn*, a corresponding task or message is introduced by *cn* in $\mathtt{net}(cn,\mathtt{rep})(cn_2)$. We already have that the remaining tasks and tasks spawned from messages in cn_1 correspond to those in cn_2. As for external messages, they are either newly introduced via the context, and are then in both composed configurations, or come from the original configuration, which we already know have coinciding external messages after applying the representation map. With respect to objects, none of the rules used in normalization change object environments, and there is a one-to-one mapping of objects between $cn_1\ cn$ and $\mathtt{net}(cn,\mathtt{rep})(cn_2)$, so after running Algorithm 2, all object containers will coincide. Hence, $\mathtt{net}(cn_1\ cn) \equiv_2 \mathtt{net}(cn,\mathtt{rep})(cn_2)$.

For converse context closure, assume $cn_2\ cn$ is WF2, and apply *cn* to produce $\mathsf{bind}\ \overline{o}'.\mathtt{ten}(cn,\mathtt{rep}^{-1})(cn_1)$ in standard form. We first need to show that this resulting configuration is WF1. Let cn' be the multiset difference of $\mathtt{ten}(cn,\mathtt{rep}^{-1})(cn_1)$ and cn_1. For OID Uniqueness, it suffices to consider the case where $\mathtt{o}(o_1,a_1) \preceq cn_1$ and $\mathtt{o}(o_2,a_2) \preceq cn'$; then, if $o_1 = o_2$, there is a corresponding clash in in $cn_2\ cn$, violating WF2. For Task-Object Existence, assume $\mathtt{t}(o,l,s) \preceq cn'$; then, if there is no object container $\mathtt{o}(o,a) \preceq cn_1\ cn'$, this violates WF2 for the task corresponding to $\mathtt{t}(o,l,s)$ in $cn_2\ cn$.

It remains to show that $cn_2\ cn\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{o}'.\mathtt{ten}(cn,\mathtt{rep}^{-1})(cn_1)$. Again, let cn' be the multiset difference of $\mathtt{ten}(cn,\mathtt{rep}^{-1})(cn_1)$ and cn_1. The network graphs of $\mathtt{net}(cn_1\ cn')$ and $cn_2\ cn$ coincide since *cn* does not contain any nodes or links. Clearly, if and only if a task or non-external message is in *cn*, a corresponding task or message is in cn'. We already have that the remaining tasks and tasks spawned from messages in cn_1 correspond to those in cn_2. As for external messages, they are either newly introduced via the context, and are then in both composed configurations, or come from the original configuration, which we already know have coinciding external messages after applying the representation map. With respect to objects, none of the rules used in normalization change object environments, and there is a one-to-one mapping of objects between $cn_2\ cn$ and $cn_1\ cn'$, so after running Algorithm 2, objects will coincide. Hence, $\mathtt{net}(cn_1\ cn') \equiv_2 cn_2\ cn$. □

Corollary 2.8.2. *For all well-formed type 1 configurations* $\mathsf{bind}\ \overline{o}.cn$ *in standard form,* $\langle \mathtt{net}(cn) \rangle \sqsubseteq \langle \mathsf{bind}\ \overline{o}.cn \rangle$.

Proof. Suppose $\rho_1 \rightarrow \rho'_1$ for $\rho_1 = cn_1 \cdots cn_n$ and $\rho'_1 = cn_1 \cdots cn_n cn_{n+1}$. We then have $cn_n \rightarrow cn_{n+1}$, and $\rho_1 \downarrow obs$ whenever $cn_n \downarrow obs$. Let $\rho_2 = cn'_1 \cdots cn'_m$. When $cn_n = \mathtt{net}(cn)$ and $cn'_m = \mathsf{bind}\ \overline{o}.cn$, as in the present case, cn_n and cn'_m are by Theorem 2.7.4 related by some conflated witness relation $\mathcal{R}$. We use this relation when constructing the required relation on executions to qualify for inclusion in $\sqsubseteq$, by straightforwardly transferring configuration properties from $\mathcal{R}$ to executions. $\square$

Chapter 3

Efficient and Fully Abstract Routing of Futures in Object Network Overlays

Mads Dam Karl Palmskog

KTH Royal Institute of Technology, Sweden
{mfd, palmskog}@kth.se

Abstract

In distributed object systems, it is desirable to enable migration of objects between locations, e.g., in order to support efficient resource allocation. Existing approaches build complex routing infrastructures to handle object-to-object communication, typically on top of IP, using, e.g., message forwarding chains or centralized object location servers. These solutions are costly and problematic in terms of efficiency, overhead, and correctness. We show how location independent routing can be used to implement object overlays with complex messaging behavior in a sound, fully abstract, and efficient way, on top of an abstract network of processing nodes connected point-to-point by asynchronous channels. We consider a distributed object language with futures, essentially lazy return values. Futures are challenging in this context due to the strong global consistency requirements they impose. The key conclusion is that execution in a decentralized, asynchronous network can preserve the standard, network-oblivious behavior of objects with futures, in the sense of contextual equivalence. To the best of our knowledge, this is the first such result in the literature. We also believe the proposed execution model may be of interest in its own right in the context of large-scale distributed computing.

3.1 Introduction

The ability to transparently and efficiently relocate objects between processing nodes is a basic prerequisite for many tasks in large-scale distributed systems, including load balancing, resource allocation, and management. By freeing applications from the burden of resource management, they can be made simpler, more resilient, and easier to manage, resulting in lower costs for development and operation.

A central problem is how to efficiently handle object and task mobility. Since object locations change dynamically in a mobile setting, some form of application-level routing is needed for inter-object messages to reach their destinations. Various approaches have been considered in the literature; Sewell et al. [184] provide a comprehensive survey. One common implementation strategy is to use a centralized, replicated, or decentralized object location register, either for forwarding or for address lookup [63, 184, 19, 87]. This type of solution requires some form of synchronization to keep registers consistent with physical locations, or else it needs to resort to message relaying, or forwarding. Forwarding by itself is another main implementation strategy used in, e.g., the Emerald system [107], or in more recent systems like JoCaml [42]. Other solutions exist, such as broadcast or multicast search, that are useful for recovery or for service discovery [11], but hardly efficient as general purpose routing devices in large systems.

In general, we consider a mechanism for object mobility with the following properties desirable:

Low stretch In stable state, the ratio between actual and optimal route lengths (costs) should be small.

Compactness The space required at each node for storing route information should be small (sublinear in the number of destinations).

Self-stabilization Even when started in a transient state, computations should proceed correctly, and converge to a stable state. Observe that this precludes the use of locks.

Decentralization To enable scaling to large networks, routes and next-hop destinations should be computed in a decentralized fashion, at the individual nodes, and not rely on a centralized facility.

Existing solutions are quite far from meeting these requirements: location registers (centralized or decentralized) and pointer forwarding regimes both preclude low stretch, and the use of locks precludes self-stabilization.

We suggest that the root of the difficulties lies in a fundamental mismatch between the information used for search and identification (typically, object identifiers, OIDs), and the information used for routing, namely, host identifiers (typically, IP addresses). If we were to route messages not to the destination *location*, but instead to the destination *object*, it should be possible to build object network overlays which much better fit the desiderata laid out above. In earlier work, described in Chapter 2, we show that this indeed appears to be true. The key idea is to use a form of location independent (also known as *flat*, or *name independent*) routing [93, 27, 100] that allows messages to be routed directly to the destination object, independently of the physical node on which that object is currently executing. In this way, a lot of the overhead and performance constraints associated with object mobility can be eliminated, including latency and bandwidth overhead due to looking up, querying, updating, and locking object location databases, and overhead due to increased traffic for, e.g., message forwarding.

We explore this idea in the context of a simple object-based language, mABS, with asynchronous message passing and futures [32, 72, 124, 207, 153, 56]. In our context, futures can be viewed as lazy return values for method calls. The mABS language is closely related to the asynchronous fragment of the ABS (Abstract Behavioral Specification) language core [103], developed in the EU FP7 HATS project. The question we raise is how program behavior is affected by execution in a network-aware model that makes location, naming, routing and message-passing explicit, as compared to execution according to a more standard, network-oblivious "reference" semantics given in the style of rewriting logic [40]. To this end, we give a maximally nondeterministic network-aware semantics of mABS.

We allow futures to be transmitted as arguments in method calls to remote objects, which introduces complications in a distributed setting. If an object receives a placeholder, there must be some way for the object to also receive the associated value, e.g., through the caller forwarding it or through querying some lookup server. Thus, it would seem likely that location independent routing could be useful for propagation of values for futures, and as we show in this paper, indeed this is so. In order for the network-aware implementation to be correct (sound and fully abstract), we must be able to show that future assignments are unique and can propagate correctly to all objects needing the assignment. Many strategies for future propagation exist in the literature [90, 174]. In this work, we use an *eager forward* based strategy [90, 31], where values are sent along the flow of futures as soon as they are computed.

The scheme we propose assumes only a network graph with OSI layer 2 connectivity, i.e., the possibility of non-lossy, ordered communication between neighboring nodes. Such a graph can be realized in many ways—for instance, as a physical non-IP network, or as a TCP-based virtual network overlay with some desired topology.

Our main result is that the reference semantics and the network-aware semantics with eager forwarding of futures correspond in the sense of contextual equivalence [169]. To the best of our knowledge, this is the first such result in the literature, interesting in itself, as it shows that the network-aware semantics captures the abstract behavior very accurately, allowing many high-level conclusions about a program to transfer to a networked realization.

The proof of the main result relies on a normal form construction which uses two procedures. The first procedure rewrites a well-formed runtime configuration into an equivalent form where routes are optimal and messages in transit in the network have been forwarded as far as possible. The second procedure uses the first to rewrite to a form where, in addition, all object-bound messages have been received and processed and where all objects have been migrated to a single node. The correctness of the normalization process gives a Church-Rosser like property—that transitions in the network-aware semantics commute with normalization. Normalization brings configurations in the network-aware semantics close to the form of configurations in the reference semantics, allowing the argument to be completed.

The paper is organized as follows. In Section 3.3, we first introduce the mABS language syntax, and the network-oblivious reference (type 1) semantics of mABS is given in Section 3.4. In Section 3.5, we present type 1 contextual equivalence, i.e., the notion of contextual equivalence adapted to the reference semantics. Then, in Section 3.6, we turn to the network-aware (type 2) semantics and present the runtime syntax and the reduction rules, and, in Section 3.7, we adapt the contextual equivalence to this semantics. We then present the normal-form construction in Section 3.8, and complete the correctness proof in Section 3.9. We describe scheduling in Section 3.10, discuss our approach in comparison to related work in Section 3.11, and finally, in Section 3.12, we conclude. Long proofs are deferred to Section 3.13.

3.2 Notation

We sometimes use a vectorized notation to abbreviate sequences, for compactness. Thus, $\overline{x}$ abbreviates a sequence $x_1, \ldots, x_n$, possibly empty, and $x_0, \overline{x}$ abbreviates $x_0, \ldots, x_n$. Let $g : A \to B$ be a finite map. The update operation for g is $g[b/a](x) = g(x)$ if $x \neq a$ and $g[b/a](a) = b$. We use $\bot$ for bottom elements, and $A_\bot$ for the lifted set with partial order $\sqsubseteq$ such that $a \sqsubseteq b$ if and only if either $a = b \in A$ or else $a = \bot$. Also, if x is a variable ranging over A, we often use $x_\bot$ as a variable ranging over $A_\bot$. For g a function $g : A \to B_\bot$, we write $g(a) \downarrow$ if $g(a) \in B$, and $g(a) \uparrow$ if $g(a) = \bot$. The product of sets (flat CPOs) A and B is $A \times B$ with pairing (a, b) and projections π_1 and π_2.

3.3 The mABS Language

We define mABS, short for milli-ABS, a small, distributed, object-based language with asynchronous method calls and futures. Its syntax is given in Table 3.1. The mABS language extends the μABS (micro-ABS) language of message-passing processes from Chapter 2 with futures.

A program is a sequence of class definitions, appended with the variables $\overline{x}$ and a "main" statement s. The class hierarchy is flat and fixed. Classes have parameters $\overline{x}$, local variable declarations $\overline{y}$, and methods $\overline{M}$. Methods have parameters $\overline{x}$, local variable declarations $\overline{y}$ and a statement body s. We assume that variables have unique declarations. The syntax of an expression e is left open, but includes the constant **self** for referring to the current object. Expressions are required to be side-

$x, y \in \mathit{Var}$			Variable
$e \in \mathit{Exp}$			Expression
$C, m \in \mathit{SID}$			Static identifier
P	$::=$	$\overline{CL}\,\{\overline{x}, s\}$	Program
CL	$::=$	$\textbf{class } C(\overline{x})\,\{\overline{y}, \overline{M}\}$	Class definition
M	$::=$	$m(\overline{x})\,\{\overline{y}, s\}$	Method definition
s	$::=$	$s_1; s_2 \mid x = \mathit{rhs} \mid \textbf{skip} \mid \textbf{while } e\,\{s\}$	Statement
		$\mid \textbf{if } e\,\{s_1\} \textbf{ else } \{s_2\} \mid \textbf{return } e$	
rhs	$::=$	$e \mid \textbf{new } C(\overline{e}) \mid e!m(\overline{e}) \mid e.\textbf{get}$	Right-hand side

Table 3.1: mABS abstract syntax

effect free when evaluated. Statements include constructs for sequential control, as well as for asynchronous method invocation, object creation, and retrieval of values associated with futures (**get** statements).

Example 3.3.1. Suppose that `combine(hi(`v`),lo(`v`))` $=$ `process(`v`)` for integers v. In the class `Server` in the program in Listing 3.1, the method `serve` returns immediately with the result if its argument is small. Otherwise, two new servers are spawned, and the upper and lower tranches delegated to those respective servers, with the current server blocking, in turn, on the two **get** assignment statements until both results are available to combine and return. In the main block, a call to `serve` on a server object results in a future identifier, stored in the variable `fut`, which is then used to retrieve the actual result, assigned to the variable `res`. The original call spawns more server objects, which, in a network-aware implementation, can move to other nodes to balance load.

```
class Server() { ,
  serve(x) { srv1, srv2, fut1, fut2, res1, res2,
    if small(x) {
      return process(x)
    } else {
      srv1 = new Server();        srv2 = new Server();
      fut1 = srv1!serve(hi(x));   fut2 = srv2!serve(lo(x));
      res1 = fut1.get;            res2 = fut2.get;
      return combine(res1, res2)
    }
  }
}
{ srv, fut, res,
  srv = new Server(); fut = srv!serve(1537); res = fut.get
}
```

Listing 3.1: mABS server program

3.4 Reference Semantics

We first present an abstract reference semantics for mABS in the style of rewriting logic. The semantics uses a reduction relation $cn \to cn'$ where cn and cn' are *configurations*, as determined by the runtime syntax in Table 3.2. Later on, we introduce different configurations and transition relations, and so refer to configurations of "type 1" for this first semantics when we need to disambiguate. With

$x, y \in Var$			Variable
$o \in OID$			Object identifier
$p \in PVal$			Primitive value
$f \in FID$			Future identifier
$v \in Val$	$=$	$PVal \cup OID \cup FID$	Value
$z \in Name$	$=$	$OID \cup FID$	Name
$l \in TEnv$	$=$	$Var \cup \{\mathbf{ret}\} \to Val_\perp$	Task environment
$a \in OEnv$	$=$	$Var \cup \{\mathbf{self}\} \to Val_\perp$	Object environment
$tsk \in Tsk$	$::=$	$\mathsf{t}(o, l, s)$	Task
$obj \in Obj$	$::=$	$\mathsf{o}(o, a)$	Object
$fut \in Fut$	$::=$	$\mathsf{f}(f, v_\perp)$	Future
$call \in Call$	$::=$	$\mathsf{c}(o, f, m, \overline{v})$	Call
$ct \in Ct$	$::=$	$tsk \mid obj \mid call \mid fut$	Container
$cn \in Cn$	$::=$	$0 \mid ct \mid cn\ cn' \mid \mathsf{bind}\ z.cn$	Configuration

Table 3.2: mABS type 1 runtime syntax

respect to the runtime syntax, $\preceq$ is the subterm relation, and we use disjoint, denumerable sets of object identifiers $o \in OID$, future identifiers $f \in FID$, and primitive values $p \in PVal$. Values v are either primitive values, OIDs, or FIDs. Lifted values are ranged over by $v_\perp \in Val_\perp$, and we use $\sqsubseteq$ for the associated standard partial ordering. OIDs and FIDs are subject to binding similar to that in the π-calculus [144]. Later, in the type 2 semantics, this type of explicit binding is dropped. Accordingly, names are either OIDs or FIDs; we use z as a generic name variable, and names are bound using the π-like binder bind. We assume throughout that names are uniquely bound. The free names of a configuration cn is the set $\mathtt{fn}(cn)$, and $OID(cn)$ is the set of OIDs of objects occurring in cn. Standard alpha congruence applies to name binding.

Configurations are multisets of containers. Configuration juxtaposition is assumed to be commutative and associative with unit 0. In addition, we use two standard structural identities: $\mathsf{bind}\ z.0 = 0$, and, when $z \notin \mathtt{fn}(cn_2)$, $\mathsf{bind}\ z.(cn_1\ cn_2) = (\mathsf{bind}\ z.cn_1)\ cn_2$. We use a vectorized notation $\mathsf{bind}\ \overline{z}.cn$, letting $\mathsf{bind}\ \varepsilon.cn = cn$ where ε is the empty sequence. The structural identities allow us to rewrite each configuration into a *standard form* $\mathsf{bind}\ \overline{z}.cn$ such that each name in $\overline{z}$ occurs free in cn, and cn has no occurrences of the binding operator bind.

Task containers are used for method body elaboration, and future containers are used as centralized stores for assignments to futures. Task and object environments l and a, respectively, map task and object variables to values. Task environments are aware of a special variable **ret** that a task can use in order to identify its return future. Upon method invocation, a task environment is initialized using the operation $\texttt{locals}(o, f, m, \overline{v})$, which maps the formal parameters of method m in the class of o to the corresponding arguments in $\overline{v}$, initializes the method local variables to suitable null values, and maps **ret** to the return future f of the task being created. Object environments are initialized using the operation $\texttt{init}(C, \overline{v}, o)$, which maps the parameters of the class C to $\overline{v}$, **self** to o, and initializes variables as above. We use the operation $\texttt{body}(o, m)$ to retrieve the statement s in the definition of m in the class of o, and $[\![e]\!]_{(a,l)} \in \mathit{Val}$ is used for evaluating the expression e in object environment a and task environment l.

We present the reduction rules in Figure 3.1, where we assume sequential statement composition is associative with unit **skip**. The rules use the notation $cn \vdash cn' \rightarrow cn''$ as shorthand for $cn\ cn' \rightarrow cn\ cn''$.

CTXT-1: If $cn_1 \rightarrow cn_2$, then $cn \vdash cn_1 \rightarrow cn_2$

CTXT-2: If $cn_1 \rightarrow cn_2$, then $\mathsf{bind}\ z.cn_1 \rightarrow \mathsf{bind}\ z.cn_2$

WLOCAL: If $x \in \texttt{dom}(l)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, x = e; s) \rightarrow \mathsf{t}(o, l[v/x], s)$

WFIELD: If $x \in \texttt{dom}(a)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o, a)\ \mathsf{t}(o, l, x = e; s) \rightarrow \mathsf{o}(o, a[v/x])\ \mathsf{t}(o, l, s)$

SKIP: $\mathsf{t}(o, l, \mathbf{skip}; s) \rightarrow \mathsf{t}(o, l, s)$

IF-TRUE: If $[\![e]\!]_{(a,l)} \neq 0$, then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{if}\ e\,\{s_1\}\ \mathbf{else}\ \{s_2\}; s) \rightarrow \mathsf{t}(o, l, s_1; s)$

IF-FALSE: If $[\![e]\!]_{(a,l)} = 0$, then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{if}\ e\,\{s_1\}\ \mathbf{else}\ \{s_2\}; s) \rightarrow \mathsf{t}(o, l, s_2; s)$

WHILE-TRUE: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\,\{s_1\}; s) \rightarrow \mathsf{t}(o, l, s_1; \mathbf{while}\ e\,\{s_1\}; s)$

WHILE-FALSE: If $[\![e]\!]_{(a,l)} = 0$, then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\,\{s_1\}; s) \rightarrow \mathsf{t}(o, l, s)$

CALL-SEND: Let $o' = [\![e']\!]_{(a,l)}$, $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$ in
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, x = e'!m(\overline{e}); s) \rightarrow \mathsf{bind}\ f.\mathsf{t}(o, l[f/x], s)\ \mathsf{f}(f, \bot)\ \mathsf{c}(o', f, m, \overline{v})$

CALL-RCV: Let $l = \texttt{locals}(o, f, m, \overline{v})$, $s = \texttt{body}(o, m)$ in
$\mathsf{o}(o, a) \vdash \mathsf{c}(o, f, m, \overline{v}) \rightarrow \mathsf{t}(o, l, s)$

RET: Let $f = l(\mathbf{ret})$, $v = [\![e]\!]_{(a,l)}$ in $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{return}\ e; s)\ \mathsf{f}(f, \bot) \rightarrow \mathsf{f}(f, v)$

GET: Let $f = [\![e]\!]_{(a,l)}$ in $\mathsf{o}(o, a)\ \mathsf{f}(f, v) \vdash \mathsf{t}(o, l, x = e.\mathbf{get}; s) \rightarrow \mathsf{t}(o, l[v/x], s)$

NEW: Let $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \texttt{init}(C, \overline{v}, o')$ in
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \rightarrow \mathsf{bind}\ o'.\mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a')$

Figure 3.1: mABS type 1 reduction rules

A method call causes a new future identifier to be created, along with its future container with lifted value initialized to $\bot$. Future instantiation is done when **return** statements are evaluated, and **get** assignment statements cause the evaluating task to hang until the value associated with the future is defined, and then store that value. Object creation (**new**) statements cause new objects to be created along with their OIDs.

We note some basic properties of the reduction semantics.

Proposition 3.4.1. *Suppose $cn \to cn'$. Then, the following holds:*

1. $\mathtt{fn}(cn') \subseteq \mathtt{fn}(cn)$.
2. *If* $\mathsf{o}(o, a) \preceq cn$, *then* $\mathsf{o}(o, a') \preceq cn'$ *for some object environment* a'.
3. *If* $\mathsf{f}(f, v_\bot) \preceq cn$, *then* $\mathsf{f}(f, v'_\bot) \preceq cn'$ *for some* $v'_\bot$ *such that* $v_\bot \sqsubseteq v'_\bot$.

Proof. No structural identity, nor any reduction rule, allows an OID or FID to escape its binder. No rules allow object or future containers to be removed. Also, no rules allow futures to be re-instantiated to $\bot$. The results follow. $\Box$

Executions of programs in the semantics are sequences of configurations derived by the rules, starting from an initial configuration, as defined below.

Definition 3.4.2 (Type 1 Initial Configuration)**.** Consider a program $\overline{CL}\,\{\overline{x}, s\}$. Assume a reserved OID o_{main}, and a reserved FID f_{init}. A *type 1 initial configuration* for the program is a configuration cn_{init} of the shape

$$\mathsf{bind}\ o_{main}, f_{init}.\mathsf{o}(o_{main}, \bot)\ \mathsf{t}(o_{main}, l_{init}, s)\ \mathsf{f}(f_{init}, \bot)$$

where l_{init} is the initial task environment assigning suitable default values to the variables in $\overline{x}$, and $l_{init}(\mathbf{ret}) = f_{init}$.

We next define a set of conditions for determining whether a configuration is "reasonable", and thus will not behave in an unexpected way under the rules, after first making precise when a future f is active for some object. Intuitively, f is active for o when f can at some point be used in a **get** statement one of o's tasks. For this to be the case, f must either be stored in some environment related to o, occur in a call container addressed to o, or be the value of some other future that is active for o.

Definition 3.4.3 (Type 1 Active Future)**.** Let cn be a type 1 configuration. Inductively, the future identifier f is *active* for the object with identifier o in cn if one of the following holds:

1. There is an object container $\mathsf{o}(o, a) \preceq cn$ such that $a(x) = f$ for some x.
2. There is a task container $\mathsf{t}(o, l, s) \preceq cn$ such that $l(x) = f$ for some x.

3. There is a call container $\mathsf{c}(o, f', m, \overline{v}) \preceq cn$, and $f' = f$ or f occurs in $\overline{v}$.

4. There is a future identifier f' that is active for o in cn, and $\mathsf{f}(f', f) \preceq cn$.

Definition 3.4.4 (Type 1 Well-Formedness)**.** A configuration cn is *type 1 well-formed* (WF1) if cn satisfies:

1. *OID Uniqueness*: If $\mathsf{o}(o_1, a_1)$ and $\mathsf{o}(o_2, a_2)$ are distinct object container occurrences in cn, then $o_1 \neq o_2$.

2. *Task-Object Existence*: If $\mathsf{t}(o, l, s) \preceq cn$, then $\mathsf{o}(o, a) \preceq cn$ for some object environment a.

3. *Call Uniqueness*: If $\mathsf{c}(o_1, f_1, m_1, \overline{v}_1)$ and $\mathsf{c}(o_2, f_2, m_2, \overline{v}_2)$ are distinct call container occurrences in cn, then $f_1 \neq f_2$.

4. *Future Uniqueness*: If $\mathsf{f}(f_1, v_{\perp,1})$ and $\mathsf{f}(f_2, v_{\perp,2})$ are distinct future container occurrences in cn, then $f_1 \neq f_2$.

5. *Single Writer*: If $\mathsf{t}(o_1, l_1, s_1)$ and $\mathsf{t}(o_2, l_2, s_2)$ are distinct task container occurrences in cn such that $l_1(\mathbf{ret}) = f_1$ and $l_2(\mathbf{ret}) = f_2$, then $f_1 \neq f_2$, $\mathsf{f}(f_1, \perp) \preceq cn$, and $\mathsf{f}(f_2, \perp) \preceq cn$, and additionally, if $\mathsf{c}(o, f, m, \overline{v}) \preceq cn$ for some o, f, m, and $\overline{v}$, then $f \neq f_1$ and $f \neq f_2$.

6. *Future Existence*: If f is active for o in cn or $\mathsf{f}(f', f) \preceq cn$, then $\mathsf{f}(f, v_\perp) \preceq cn$; if $\mathsf{f}(f, \perp) \preceq cn$, then there is either a call container $\mathsf{c}(o', f, m, \overline{v}) \preceq cn$ with $o' \in \mathit{OID}(cn)$, or a task container $\mathsf{t}(o', l, s) \preceq cn$ with $l(\mathbf{ret}) = f$.

Well-formedness is important, as it ensures that objects, calls, and futures are defined uniquely (OID Uniqueness, Future Uniqueness, Call Uniqueness), and that tasks are defined only along with their accompanying object (Task-Object Existence). The Single Writer property ensures that only the task that was spawned along with some given future is able to assign to that future, and hence, if the task has not yet returned, the future remains uninstantiated. Future Existence ensures that whenever an object has access to an FID, there exists a corresponding future container which either already contains a value or has the potential to contain one further on in an execution.

Proposition 3.4.5 (WF1 Preservation)**.** *Let cn be a configuration. Then, the following holds:*

1. *If cn is a type 1 initial configuration, then cn is WF1.*

2. *If cn is WF1 and $cn \to cn'$, then cn' is WF1.*

Proof. By inspection of the definitions and the rules. □

3.5 Type 1 Contextual Equivalence

Our approach to implementation correctness uses contextual equivalence [169], which requires of a pair of equivalent configurations, firstly, that the internal transition relation $\rightarrow$ is preserved in both directions, and secondly, that the relation is preserved when adding a context configuration, all while preserving a set of external observations. A number of works [101, 180] have established strong relations between contextual equivalence for reduction oriented semantics and bisimulation/-logical relation based equivalences for sequential and higher-order computational models.

Assume an OID *ext* representing the "outside world", not allowed to be bound or defined in any well-formed configuration. An observation, or *barb*, is a call to *ext* with evaluated arguments, of the form $ext!m(\overline{v})$, ranged over by *obs*. The observation predicate $cn \downarrow obs$ is defined to hold just in case we have $cn = \mathsf{bind}\ \overline{z}.cn'\ \mathsf{c}(ext, f, m, \overline{v})$ for some $\overline{z}$, cn', and f. The derived predicate $cn \Downarrow obs$ holds just in case $cn \rightarrow^* cn'' \downarrow obs$ for some cn''.

Definition 3.5.1 (Type 1 Witness Relation, Type 1 Contextual Equivalence)**.** Let $\mathcal{R}$ range over binary relations on WF1 configurations. The relation $\mathcal{R}$ is a *type 1 witness relation*, if $cn_1\ \mathcal{R}\ cn_2$ implies

1. *Reduction Closure*: If $cn_1 \rightarrow cn_1'$, then $cn_2 \rightarrow^* cn_2'$ for some cn_2' such that $cn_1'\ \mathcal{R}\ cn_2'$.
2. *Context Closure*: If $cn_1\ cn$ is WF1, then $cn_2\ cn$ is WF1 and $cn_1\ cn\ \mathcal{R}\ cn_2\ cn$.
3. *Barb Preservation*: If $cn_1 \downarrow obs$, then $cn_2 \Downarrow obs$.

Additionally, the converse properties must hold with $\mathcal{R}^{-1}$ for $\mathcal{R}$ above. We define *type 1 contextual equivalence*, $\simeq_1$, as the union of all type 1 witness relations. Additionally, we say that the WF1 configurations cn_1 and cn_2 are *type 1 contextually equivalent* whenever $cn_1 \simeq_1 cn_2$, i.e., whenever $cn_1\ \mathcal{R}\ cn_2$ for some type 1 witness relation $\mathcal{R}$.

We establish some well-known, elementary properties of contextual equivalence for later reference.

Proposition 3.5.2. *The identity relation is a type 1 witness relation. $\simeq_1$ is a type 1 witness relation. If $\mathcal{R}$, $\mathcal{R}_1$, and $\mathcal{R}_2$ are type 1 witness relations then so are*

1. $\mathcal{R}^{-1}$*,*

2. $\mathcal{R}^*$*, and*

3. $\mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1$*.*

Proof. See Section 3.13. □

Proposition 3.5.3. $\simeq_1$ *is an equivalence relation.*

Proof. The result follows from Proposition 3.5.2; for transitivity, in particular, we use property 3. □

3.6 Network-Aware Semantics

We now address the problem of efficiently executing mABS programs on an abstract network graph using the location independent routing scheme alluded to in Section 3.1. In addition to the naming, routing, and object migration issues already addressed previously in Chapter 2, the additional challenge is to ensure that futures are correctly assigned and propagated at the network level.

In the network-aware semantics, we assume an explicitly given network of nodes and directional links with which message buffers are associated, modeling a concrete network structure with asynchronous point-to-point message passing. Object execution is localized to each node. As routing information is propagated, inter-node object-to-object message delivery becomes possible. Objects can migrate between neighboring nodes. The propagation of routing information will automatically lead to routing tables becoming up-to-date. Method calls to an object can be issued if a task can access the OID of the object, and the associated messages can be delivered once a route to the callee becomes known.

We use an eager forward based strategy for handling future propagation. The central idea of the strategy is that, whenever an object shares a future identifier, the object assumes an obligation to send the associated value to the object with which the future identifier is shared. The value of the shared future may be unavailable, requiring the use of forwarding lists for futures, stored in the object state. Our objective is to prove that this approach is sound and fully abstract for our network-aware semantics, even though routing may be in an unstable state.

Example 3.6.1. A fragment of an execution of a program in the network-aware semantics, which illustrates future propagation and its interaction with routing, is shown in Figure 3.2. In the configuration in Figure 3.2(a), a call to method m with argument f, a future identifier, is about to be sent from object o_0 residing on node u_0 to object o_1 on node u_2. The following changes then take place as the system evolves into the configuration Figure 3.2(b):

- A new future f' is created at object o_0, as a placeholder for the return value of the call to method m.
- The forwarding list for f at o_0 is augmented to include o_1; a dashed arrow indicates this in the figure.
- The call is routed to o_1, where a new task computing the statement $\texttt{body}(o_1, m)$ is spawned, associated with f'.

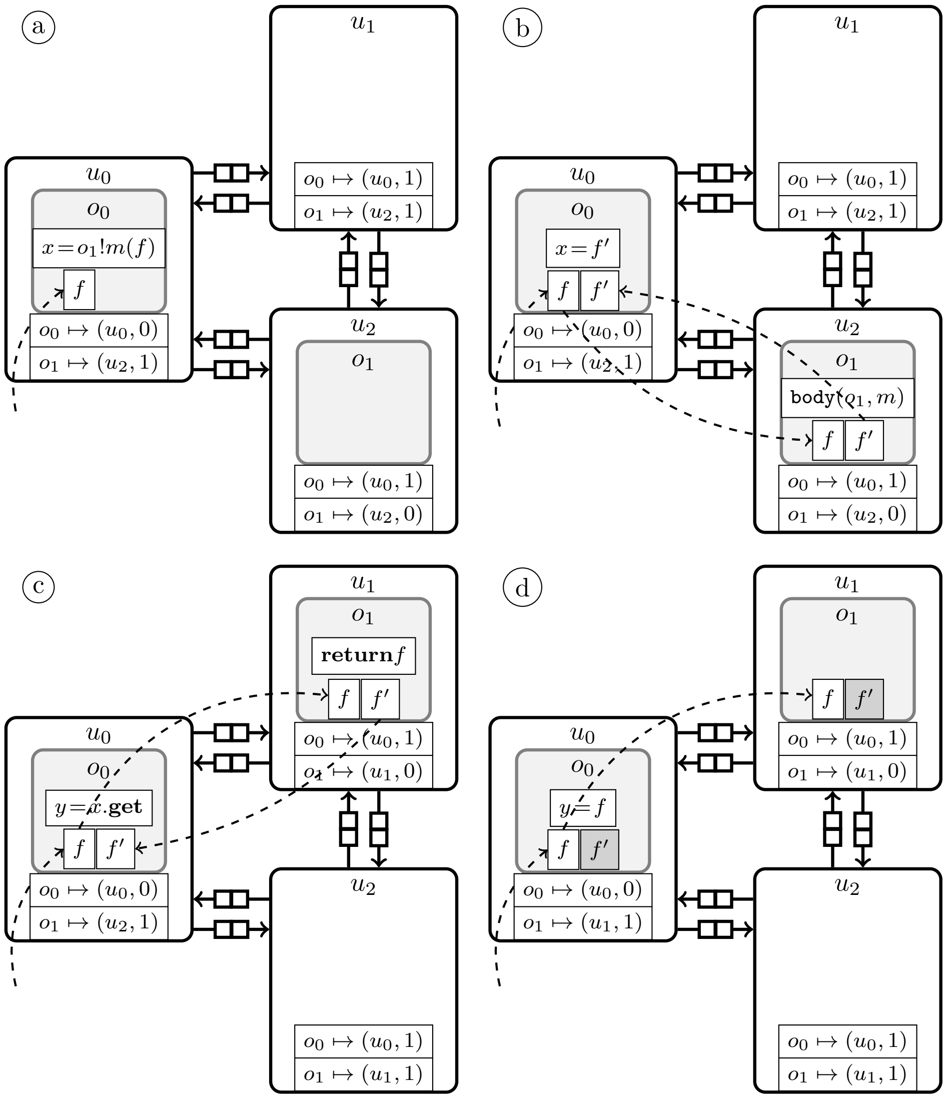


Figure 3.2: Execution fragment with future forwarding in the type 2 semantics

- o_1 is augmented with f and f' as placeholders, and the forwarding list for f' at o_1 is augmented to include o_0.

The scheduler now decides to migrate o_1 from u_2 to u_1. No action is required other than regular routing table exchanges, as the forwarding lists keep referencing the same objects. Finally, the future identifier f is returned by the task at o_1, as shown in the configuration in Figure 3.2(c). Note that routing information has not stabilized at this point; the next hop to o_1 at u_0 is still u_2 and not u_1. In the configuration in Figure 3.2(d), the value of the future f', namely, f, has been sent to o_0, where it becomes assigned to the variable y. Routing tables are now stable. A dark grey box for a future in an object indicates that the future is resolved at that object. The obligation of o_0 to forward the value of f to o_1 has become redundant and could be removed without affecting object behavior, but for simplicity we do not include such garbage collection in the semantics. The price for this omission is additional messaging, but the latency remains the same, since o_0 is able to use an assignment to f when it is first received there.

3.6.1 Runtime Syntax

In Table 3.3, we present the network-aware mABS runtime syntax, i.e., the shape of the runtime state. We adopt the same syntactical conventions as in Section 3.4 and use indices to disambiguate. For instance, Obj_1 is the set Obj of the type 1 semantics in Table 3.2, and Obj_2 is the corresponding set in Table 3.3. Tasks are unchanged, and we write $\mathsf{t}(cn)$ for the multiset of tasks in cn, i.e., the multiset $\{tsk \mid tsk \preceq cn\}$, and $\mathsf{o}(cn)$ for the multiset of objects in cn, similarly defined. We also write $\mathsf{m}(cn)$ for the multiset $\{msg \mid msg \preceq cn\}$.

$u \in NID$			Node identifier
$t \in RTable$	$=$	$OID \rightarrow (NID \times \omega)_\perp$	Routing table
$q \in Q$	$=$	Msg^*	Queue
$a \in OEnv_2$	$=$	$(Var \cup \{\mathbf{self}\} \rightarrow Val_\perp) \times$ $(FID \rightarrow (Val_\perp \times (OID \text{ list}))_\perp)$	Object environment
$obj \in Obj_2$	$::=$	$\mathsf{o}(o, a, u, q_{in}, q_{out})$	Object
$nd \in Nd$	$::=$	$\mathsf{n}(u, t)$	Node
$lnk \in Lnk$	$::=$	$\mathsf{l}(u, q, u')$	Link
$ct \in Ct_2$	$::=$	$tsk \mid obj \mid nd \mid lnk$	Container
$cn \in Cn_2$	$::=$	$ct_1 \ldots ct_n$	Configuration
$msg \in Msg$	$::=$	$\mathsf{call}(o, o', f, m, \overline{v}) \mid \mathsf{table}(t)$ $\mid \mathsf{object}(cn) \mid \mathsf{future}(o, f, v)$	Message

Table 3.3: mABS type 2 runtime syntax

Network and Routing The nodes and links in a configuration cn induce a network graph $\mathtt{graph}(cn)$, which contains a vertex u for each node container $\mathsf{n}(u, t)$ and an edge (u, u') for each link $\mathsf{l}(u, q, u')$. The reduction semantics given later does not allow identifiers in nodes or links to be changed, so in the context of any given transition (or, execution), the network graph remains constant. Note that there is no a priori guarantee that the network graph is a well-formed, which could lead to unexpected behavior in an execution. Subsequently, we therefore impose some constraints on the well-formedness of the network graph of a configuration.

Definition 3.6.2 (Network Graph Well-formedness)**.** A configuration cn has a *well-formed network graph* if it satisfies:

1. *Vertex Existence*: $\mathsf{n}(u, t) \preceq cn$ for some u and t.
2. *Edge Endpoint Existence*: If we have $\mathsf{l}(u_1, q, u_2) \preceq cn$, then $\mathsf{n}(u_1, t_1) \preceq cn$ and $\mathsf{n}(u_2, t_2) \preceq cn$.
3. *Unique Vertices*: If $\mathsf{n}(u_1, t_1)$ and $\mathsf{n}(u_2, t_2)$ are distinct occurrences in cn, then $u_1 \neq u_2$.
4. *Unique Edges*: If $\mathsf{l}(u_1, q_1, u_1')$ and $\mathsf{l}(u_2, q_2, u_2')$ are distinct occurrences in cn, then $u_1 \neq u_2$, or $u_1' \neq u_2'$, or both.
5. *Reflexivity*: If $\mathsf{n}(u, t) \preceq cn$, then $\mathsf{l}(u, q, u) \preceq cn$.
6. *Symmetry*: If we have $\mathsf{n}(u_1, t_1) \preceq cn$, $\mathsf{n}(u_2, t_2) \preceq cn$, and $\mathsf{l}(u_1, q, u_2) \preceq cn$, then $\mathsf{l}(u_2, q', u_1) \preceq cn$.
7. *Connectedness*: If $\mathsf{n}(u_1, t_1) \preceq cn$ and $\mathsf{n}(u_2, t_2) \preceq cn$, then there is a path from u_1 to u_2 in $\mathtt{graph}(cn)$.

Vertex Existence rules out uninteresting networks where no node is available to execute tasks. Edge Endpoint Existence ensures messages on links have the possibility of being received by a node on the other side. Unique Vertices and Unique Edges avoids duplicate network entities, which could irrevocably confuse the routing system and prevent progress. Reflexivity ensures the existence of a default route for otherwise unroutable messages. Symmetry makes mutual exchanges of routing tables possible between nodes. Connectedness rules out lack of progress due to the location of an object being unreachable from the location of a message addressed to that object.

For routing, we adopt a rudimentary Bellman-Ford distance vector discipline [193]; better and more complex routing schemes can be used without affecting the results. For a routing table t, $t(o) = (u, n)$ indicates that, as far as t is concerned, there is a path from the current node (the node to which t is attached) to the object o with distance n, that first visits the node u. For simplicity, we only count hops to compute distance. Next hop lookup is performed by the operation $\mathtt{nxt}$, where $\mathtt{nxt}(o, t) = \pi_1(t(o))$. There is also an operation $\mathtt{upd}$ for updating a routing table t by a routing table t' received from a neighboring node u, defined by

$$\mathtt{upd}(t, u, t')(o) = \begin{cases} \bot & \text{if } o \notin \mathtt{dom}(t) \cup \mathtt{dom}(t') \\ t(o) & \text{else, if } o \notin \mathtt{dom}(t') \\ (u, \pi_2(t'(o)) + 1) & \text{else, if } o \notin \mathtt{dom}(t) \text{ or } \pi_1(t'(o)) = u \\ (u, \pi_2(t'(o)) + 1) & \text{else, if } \pi_2(t'(o)) + 1 < \pi_2(t(o)) \\ t(o) & \text{otherwise.} \end{cases}$$

Finally, there is an operation $\texttt{reg}(o, u, t, n)$ that returns the routing table t', obtained by registering the object identifier o at t's current node u with distance n, i.e., such that

$$\texttt{reg}(o, u, t, n)(o') = \begin{cases} (u, n) & \text{if } o = o' \\ t(o') & \text{otherwise.} \end{cases}$$

Message Queues and Messages FIFO message queue operations are standard: $\texttt{hd}(q)$ returns the head of q, while $\texttt{enq}(msg, q)$ enqueues a message msg onto the tail of q, and $\texttt{deq}(q)$ returns q with $\texttt{hd}(q)$ removed. If q is empty, then $\texttt{hd}(q) = \texttt{deq}(q) = \bot$. Messages in queues have the following forms:

- $\mathsf{call}(o, o', f, m, \overline{v})$ corresponds to a call container in the type 1 semantics, with o the identifier of the callee and o' the identifier of the caller, respectively. The identifier o' is needed to enable forwarding the resolved value of f to the caller.

- $\mathsf{future}(o, f, v)$ informs the object o that the future f has been instantiated to the value v.

- $\mathsf{object}(cn)$ is used for migrating objects and their tasks across nodes. The configuration cn is the *closure* of a specific object, as explained below.

- $\mathsf{table}(t)$ carries the routing table t from one node to another.

Call and future messages are said to be *object bound*, while table messages and object closure messages are *node bound*. We define $\texttt{dst}(msg)$, the *destination* of msg, to be o for a call message or a future message as defined above, and $\bot$ in the remaining two cases.

Objects and Object Environments Object containers $\mathsf{o}(o, a, u, q_{in}, q_{out})$ are attached to a node u and equipped with an ingoing (q_{in}) and an outgoing (q_{out}) FIFO message queue, and object environments a are augmented with a mapping of futures f to pairs $(v_\bot, \overline{o})$, where $v_\bot$ is the lifted value currently associated with f at the current object, and $\overline{o}$ is a *forwarding list*, containing the identifiers of the objects subscribing to instantiations of f at the object. For instance, if $\pi_2(a)(f) = (\bot, o_1\, o_2)$, the future f is as yet uninstantiated (at the object to which a belongs), and, if f eventually does become instantiated, the instantiation must be forwarded in a future message to o_1 and o_2. Forwarding does not necessarily happen in the given order, since we consider forwarding lists modulo associativity and commutativity with the empty list ε as unit. We use the following notation and auxiliary operations related to object environments:

- $a(x)$ abbreviates $\pi_1(a)(x)$ and $a(f)$ abbreviates $\pi_2(a)(f)$.

- $a[v/x]$ is a with $\pi_1(a)$ replaced by the expected update.

- $a[v/f]$ updates $\pi_2(a)$ by mapping f to $(v, \pi_2(a(f)))$, i.e., the assigned value is updated and the forwarding list is unchanged; if $f \notin \mathtt{dom}(\pi_2(a))$, then $a[v/f](f) = (v, \varepsilon)$.

- $a[(v, \overline{o})/f]$ performs the expected update where both the value and the forwarding list are changed.

- $\mathtt{fw}(\overline{v}, o, a)$ updates $\pi_2(a)$ by, for each future f occurring in $\overline{v}$, adding o to the forwarding list of $a(f)$, i.e., by mapping f to either $(\bot, o)$ if $a(f) \uparrow$, or $(\pi_1(a(f)), o\,\pi_2(a(f)))$ otherwise.

- $\mathtt{init}(C, \overline{v}, o)$ returns an initial object environment, by mapping the formal parameters of C to $\overline{v}$, and **self** to o, as in the type 1 semantics.

- $\mathtt{init}(\overline{v}, a)$ augments a by mapping each FID f in $\overline{v}$ which is uninitialized in a (i.e., $a(f) \uparrow$) to $(\bot, \varepsilon)$.

As a consequence of these changes, futures are eliminated as containers in the type 2 runtime syntax. In many other respects, the syntax is unchanged: syntactical conventions that are not modified from the type 1 runtime syntax above remain the same. In particular, we continue to assume the commutativity and associativity properties of configuration juxtaposition, now with the empty container list as unit.

Object Closures For an object message $\mathsf{object}(cn)$ to be valid, the configuration cn needs to be an *object closure* which wraps an object container with all its tasks. For example, if the object o has precisely n tasks, its closure has the form

$$\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l_1, s_1)\ \ldots\ \mathsf{t}(o, l_n, s_n).$$

Note that containers inside object closures are included in the subterm relation $\preceq$ for configurations where the object message resides. We use the following auxiliary operations to handle closures:

- $\mathtt{clo}(cn, o)$ is the closure of object o with respect to the configuration cn, namely, the multiset of all type 2 containers of the form $\mathsf{o}(o', a', u', q'_{in}, q'_{out})$ or $\mathsf{t}(o', l', s')$ in cn, such that $o' = o$.

- $\mathtt{oidof}(cn)$ returns the OID o, if all the type 2 containers in cn are objects and tasks with OID o.

- $\mathtt{place}(cn, u)$ places all object containers in the configuration cn at the node u, i.e., cn and $\mathtt{place}(cn, u)$ are identical, except that object containers have their NID replaced with u.

3.6.2 Reduction Semantics

An important distinction between the reference semantics and the network-aware semantics is the absence of binding. For the standard semantics, name binding allows us to avoid clashes between locally generated names. However, since all name generation in the mABS type 2 semantics takes place in the context of a given NID u, we can simply assume the existence of two operations $\mathtt{newf}(u)$ and $\mathtt{newo}(u)$, which return a new future identifier and a new object identifier, respectively, that is globally fresh for the "current context", with $\mathtt{newo}(u)$ distinct from *ext*.

We now present the mABS type 2 reduction rules. The first part, shown in Figure 3.3, is carried over from the type 1 semantics in Figure 3.1, with some minor modifications. First, CTXT-2 is dropped, since name binding is dropped from the type 2 runtime syntax. Second, WLOCAL, WFIELD, IF-TRUE, IF-FALSE, WHILE-TRUE, and WHILE-FALSE are straightforwardly modified to account for the new runtime shape of objects. The remaining reduction rules are given in Figure 3.4 and Figure 3.5; these rules can be divided into groups as per below.

CTXT-1: If $cn_1 \to cn_2$, then $cn \vdash cn_1 \to cn_2$

WLOCAL-2: If $x \in \mathtt{dom}(l)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, x = e; s) \to \mathsf{t}(o, l[v/x], s)$

WFIELD-2: If $x \in \mathtt{dom}(a)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$

SKIP: $\mathsf{t}(o, l, \textbf{skip}; s) \to \mathsf{t}(o, l, s)$

IF-TRUE-2: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, \textbf{if}\ e\,\{s_1\}\ \textbf{else}\ \{s_2\}; s) \to \mathsf{t}(o, l, s_1; s)$

IF-FALSE-2: If $[\![e]\!]_{(a,l)} = 0$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, \textbf{if}\ e\,\{s_1\}\ \textbf{else}\ \{s_2\}; s) \to \mathsf{t}(o, l, s_2; s)$

WHILE-TRUE-2: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, \textbf{while}\ e\,\{s_1\}; s) \to \mathsf{t}(o, l, s_1; \textbf{while}\ e\,\{s_1\}; s)$

WHILE-FALSE-2: If $[\![e]\!]_{(a,l)} = 0$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, \textbf{while}\ e\,\{s_1\}; s) \to \mathsf{t}(o, l, s)$

Figure 3.3: mABS type 2 reduction rules, part 1

Message Passing T-SEND and T-RCV are concerned with the exchange of routing tables, which only takes place between distinct adjacent nodes. MSG-SEND, MSG-RCV, and MSG-ROUTE are used to manage message passing, i.e., reading a message from a link queue and transferring it to the appropriate object in-queue, and dually, reading a message from an out-queue and transferring it to the attached link queue. If the destination object does not reside at the current node, the message is routed

T-SEND: If $u \neq u'$, then $\mathsf{n}(u,t) \vdash \mathsf{l}(u,q,u') \to \mathsf{l}(u, \mathtt{enq}(\mathsf{table}(t), q), u')$

T-RCV: If $\mathtt{hd}(q) = \mathsf{table}(t')$, then
$\mathsf{l}(u',q,u)\ \mathsf{n}(u,t) \to \mathsf{l}(u', \mathtt{deq}(q), u)\ \mathsf{n}(u, \mathtt{upd}(t,u',t'))$

MSG-SEND: If $\mathtt{hd}(q_{out}) = msg$, $\mathtt{dst}(msg) = o'$, and $\mathtt{nxt}(o',t) = u'$, then
$\mathsf{n}(u,t) \vdash \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{l}(u,q,u') \to$
$\mathsf{o}(o,a,u,q_{in},\mathtt{deq}(q_{out}))\ \mathsf{l}(u, \mathtt{enq}(msg,q), u')$

MSG-RCV: If $\mathtt{hd}(q) = msg$ and $\mathtt{dst}(msg) = o$, then
$\mathsf{l}(u',q,u)\ \mathsf{o}(o,a,u,q_{in},q_{out}) \to \mathsf{l}(u',\mathtt{deq}(q),u)\ \mathsf{o}(o,a,u,\mathtt{enq}(msg,q_{in}),q_{out})$

MSG-ROUTE: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, $\mathtt{nxt}(o,t) = u''$, and $u'' \neq u$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u',q,u)\ \mathsf{l}(u,q',u'') \to \mathsf{l}(u',\mathtt{deq}(q),u)\ \mathsf{l}(u,\mathtt{enq}(msg,q'),u'')$

MSG-DELAY-1: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, and $\mathtt{nxt}(o,t)\uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u',q,u)\ \mathsf{l}(u,q',u) \to \mathsf{l}(u',\mathtt{deq}(q),u)\ \mathsf{l}(u,\mathtt{enq}(msg,q'),u)$

MSG-DELAY-2: If $\mathtt{hd}(q_{out}) = msg$, $\mathtt{dst}(msg) = o'$, and $\mathtt{nxt}(o',t)\uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{l}(u,q,u) \to$
$\mathsf{o}(o,a,u,q_{in},\mathtt{deq}(q_{out}))\ \mathsf{l}(u,\mathtt{enq}(msg,q),u)$

MSG-DELAY-3: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, and $\mathtt{nxt}(o,t)\uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u,q,u) \to \mathsf{l}(u,\mathtt{enq}(msg,\mathtt{deq}(q)),u)$

Figure 3.4: mABS type 2 reduction rules, part 2

to the next link. In MSG-RCV, note that the receiving node is not required to be present. However, its existence is enforced by the well-formedness condition for the network graph.

Unstable Routing MSG-DELAY-1, MSG-DELAY-2, and MSG-DELAY-3 are used to handle the cases where routing tables have not yet stabilized, or a message is unroutable. For instance, it may happen that updates to the routing tables have not yet caught up with object migration. In this case, a message may enter an object out-queue without the hosting node's routing table having information about the message's destination (MSG-DELAY-2). Another case is when a node receives a message on a link without knowing where to forward it (MSG-DELAY-1). This situation is particularly problematic, as a blocked message may prevent routing table updates to reach the hosting node, thus causing a deadlock. The solution we propose, which is implicit in the rules, is to use the network self-loop links, included in all well-formed networks, as buffers for unroutable messages that may or may not become routable. MSG-DELAY-3 allows messages on this link to be shuffled.

Inter-Object Messaging CALL-SEND-2, CALL-RCV-2, FUT-SEND, and FUT-RCV produce and consume method call and future instantiation messages, respectively. A method call causes a local future identifier to be created and passed along with the

CALL-SEND-2: Let $o' = [\![e']\!]_{(a,l)}$, $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $f = \texttt{newf}(u)$,
$a' = \texttt{fw}(\overline{v}, o', \texttt{init}(f, a))$ in $\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e'!m(\overline{e}); s) \rightarrow$
$\mathsf{o}(o, a', u, q_{in}, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), q_{out}))\ \mathsf{t}(o, l[f/x], s)$

CALL-RCV-2: If $\texttt{hd}(q_{in}) = \mathsf{call}(o, o', f, m, \overline{v})$, then
let $a' = \texttt{fw}(f, o', \texttt{init}(\overline{v}, a))$, $l = \texttt{locals}(o, f, m, \overline{v})$, $s = \texttt{body}(o, m)$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \rightarrow \mathsf{o}(o, a', u, \mathsf{deq}(q_{in}), q_{out})\ \mathsf{t}(o, l, s)$

FUT-SEND: If $a(f) = (v, o_1\, \overline{o_2})$, then let $a' = \texttt{fw}(v, o_1, a[(v, \overline{o_2})/f])$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \rightarrow \mathsf{o}(o, a', u, q_{in}, \mathsf{enq}(\mathsf{future}(o_1, f, v)), q_{out})$

FUT-RCV: If $\texttt{hd}(q_{in}) = \mathsf{future}(o, f, v)$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \rightarrow \mathsf{o}(o, a[v/f], u, \mathsf{deq}(q_{in}), q_{out})$

RET-2: Let $v = [\![e]\!]_{(a,l)}$, $f = l(\mathbf{ret})$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, \mathbf{return}\ e; s) \rightarrow \mathsf{o}(o, a[v/f], u, q_{in}, q_{out})$

GET-2: If $[\![e]\!]_{(a,l)} = f$ and $\pi_1(a(f)) = v$, then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{t}(o, l, x = e.\mathbf{get}; s) \rightarrow \mathsf{t}(o, l[v/x], s)$

NEW-2: Let $o' = \texttt{newo}(u)$, $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \texttt{fw}(\overline{v}, o', a)$,
$a'' = \texttt{init}(\overline{v}, \texttt{init}(C, \overline{v}, o'))$ in $\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \rightarrow$
$\mathsf{o}(o, a', u, q_{in}, q_{out})\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a'', u, \varepsilon, \varepsilon)$

OBJ-REG: $\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{n}(u, t) \rightarrow \mathsf{n}(u, \texttt{reg}(o, u, t, 0))$

OBJ-SEND: If $u \neq u'$, then let $t' = \texttt{reg}(o, u', t, 1)$, $cn' = \texttt{clo}(cn, o)$ in
$\mathsf{n}(u, t)\ \mathsf{l}(u, q, u')\ cn \rightarrow \mathsf{n}(u, t')\ \mathsf{l}(u, \mathsf{enq}(\mathsf{object}(cn'), q), u')\ (cn - cn')$

OBJ-RCV: If $\texttt{hd}(q) = \mathsf{object}(cn)$, then
$\mathsf{l}(u', q, u)\ \mathsf{n}(u, t) \rightarrow \mathsf{l}(u', \mathsf{deq}(q), u)\ \mathsf{n}(u, \texttt{reg}(\texttt{oidof}(cn), u, t, 0))\ \texttt{place}(cn, u)$

Figure 3.5: mABS type 2 reduction rules, part 3

call message. Upon receiving the call, the callee first initializes the received futures it does not already know about, and then augments the resulting local object environment to enact forwarding for the received return future to the caller, when this becomes possible. The eventual return value becomes associated with the return future by the mapping to the constant **ret** during initialization of the task's local environment. FUT-SEND lets future instantiations be forwarded to objects in the forwarding list whenever the future is instantiated to a value locally, and FUT-RCV causes the receiving object to update its local environment accordingly. A future may itself be instantiated to a future, making it necessary to update the local forwarding list whenever FUT-SEND is used.

Language Constructs RET-2, GET-2, and NEW-2 handle the corresponding language constructs. Return statements cause the corresponding future to be instantiated, as explained above. As expected, **get** statements read the value of the future, provided it has received a value, and **new** statements cause a new object to be created, initialized, and registered at the local node. The arguments provided to the new object may contain future identifiers, whose values must be duly forwarded to the new object by augmenting the preexisting object's forwarding lists.

Object Registration and Migration OBJ-REG registers a new object on the node on which it has been placed. The final rules concern object migration. Of these, the rule OBJ-SEND is *global* in that it is not allowed to be used in subsequent applications of the CTXT-1 rule. In this way, we can guarantee that only complete object closures are migrated. To remove an object closure cn' from a configuration cn for migration, we take the multiset difference $cn - cn'$.

All of the above rules are strictly local and appeal only to mechanisms directly implementable at the link level, i.e., they correspond to tests and simple datatype manipulations taking place at a single node, or accesses to a single node's link layer interface. The "global" property appealed to above for migration is merely a formal device to enable a convenient treatment of object closures—an object and all its tasks will always be co-located.

The reduction rules can be optimized in several ways. For instance, object self-calls are always routed through the "network interface", i.e., the hosting node's self-loop link. This is not necessary. It would be possible to add a rule to directly spawn a handling task from a self-call without affecting the results of the paper.

We note some basic properties of the network-aware semantics.

Proposition 3.6.3. *Suppose that $cn \to cn'$. Then, the following holds:*

1. *If $\mathsf{n}(u, t) \preceq cn$, then $\mathsf{n}(u, t') \preceq cn'$ for some t'.*
2. *If $\mathsf{l}(u, q, u') \preceq cn$, then $\mathsf{l}(u, q', u') \preceq cn'$ for some q'.*
3. *If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, then there is an object $\mathsf{o}(o, a', u', q'_{in}, q'_{out}) \preceq cn'$ (the* derivative *of the object in cn'), such that for all variables x, if $a(x)\downarrow$, then $a'(x)\downarrow$, and for all future identifiers f, if $a'(f)\downarrow$, then $a'(f)\downarrow$, and if $\pi_1(a(f))\downarrow$, then $\pi_1(a'(f))\downarrow$.*
4. *If $\mathsf{t}(o, l, s) \preceq cn$ and $l(\textbf{ret}) = f$, then either there is a task $\mathsf{t}(o, l', s') \preceq cn'$ (the* derivative *of the task in cn'), such that $\mathtt{dom}(l) \subseteq \mathtt{dom}(l')$, and $l'(\textbf{ret}) = f$, or there is an object $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn'$ such that $\pi_1(a(f))\downarrow$.*

Proof. By straightforward induction on the reduction relation. □

Initial configurations in the network-aware semantics are parameterized on a network graph configuration, but are otherwise similar to their network-oblivious counterparts.

Definition 3.6.4 (Type 2 Initial Configuration)**.** Consider a program $\overline{CL}\,\{\overline{x}, s\}$. Assume a reserved OID o_{main}, and a reserved FID f_{init}. A *type 2 initial configuration* for the program is a configuration cn_{init} of the shape

$$\mathsf{o}(o_{main}, a_{init,2}, u_{init}, \varepsilon, \varepsilon)\ \mathsf{t}(o_{main}, l_{init}, s)\ cn_{graph}$$

where

- $a_{init,2} = \bot[(\bot, \varepsilon)/f_{init}]$,
- l_{init} is unchanged from Definition 3.4.2,
- cn_{graph} is a configuration consisting only of nodes and links, inducing a well-formed network graph,
- cn_{graph} contains a node $\mathsf{n}(u_{init}, t_{init})$,
- $t_{init}(o_{main}) = (u_{init}, 0)$, and $t_{init}(o) = \bot$ for all OIDs o distinct from o_{main},
- $t(o) = \bot$ for all routing tables $t \neq t_{init}$ in cn_{graph} and for all OIDs o, and
- $\mathsf{l}(u, q, u') \preceq cn_{graph}$ implies $q = \varepsilon$, for all u and u'.

3.6.3 Type 2 Well-Formedness

Well-formedness in the case of the network-aware semantics must make sure that, e.g., multiple objects are never given identical names, and that futures are never assigned inconsistent values, as detailed below. A particularly delicate matter concerns the way future instantiations are propagated. It must be the case that either all objects that may at some time need the value of a future can also eventually receive it, or else no object is able to do so (due to task nontermination). This is the "future liveness" property in Definition 3.6.10 below.

Definition 3.6.5 (Future Assignment)**.** We say that a configuration cn *assigns* the value v *to* f if there is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, such that either $\pi_1(a(f)) = v$ or there is a message $\mathsf{future}(o, f, v) \preceq cn$. If there is no such value, we say that f is *unassigned* in cn.

Future Assignment, which is a property over all object environments, is the decentralized correlate of whether there exists a future container for f with a value v (assigned) or $\bot$ (unassigned) in the type 1 semantics.

Definition 3.6.6 (Type 2 Active Future)**.** Let cn be a type 2 configuration. Inductively, the future identifier f is *active* for the object with identifier o in cn if one of the following holds:

1. There is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ such that $a(x) = f$ for some variable x.

2. There is a task container $\mathsf{t}(o, l, s) \preceq cn$ such that $l(x) = f$ for some x.
3. There is a call message $\mathsf{call}(o, o', f', m, \overline{v})$ in transit in cn, and $f' = f$ or f occurs in $\overline{v}$.
4. There is a future identifier f' that is active for o, and cn assigns f to f'.

The changes in the notion of active future from Definition 3.4.3 are straightforward, amounting to little more than substituting runtime syntax, once we replace future container existence with Future Assignment.

Definition 3.6.7 (Notification Path)**.** Fix a type 2 configuration cn, an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ and an OID $o' \in \mathit{OID}(cn)$. Let n be a nonnegative integer. Inductively, o is *on the notification path of f by o' in n steps*, if n is the least number such that one of the following conditions hold:

1. $n = 0$, $o' = o$, and $\pi_1(a(f)) = v$.
2. $n = 1$, $o' = o$, and $\mathsf{future}(o, f, v) \preceq cn$.
3. $n = 1$, $o' = o$, and $\mathsf{t}(o, l, s) \preceq cn$ with $l(\mathbf{ret}) = f$.
4. $n = 2$, $o' = o$, and $\mathsf{call}(o, o'', f, m, \overline{v}) \preceq cn$.
5. $n = 4$, and $\mathsf{call}(o', o, f, m, \overline{v}) \preceq cn$.
6. $n = n' + 2$, and $\mathsf{o}(o'', a', u', q'_{in}, q'_{out}) \preceq cn$ such that $o \in \pi_2(a'(f))$, and o'' is on the notification path of f by o' in n' steps.
7. $n = 2n' + n''$, with n' and n'' nonnegative integers, if o is on the notification path of f' by o'' in n' steps, cn assigns f to f', and o'' is on the notification path of f by o' in n'' steps.

Say that o is *on the notification path of f*, if o is on the notification path of f by some o' in some number of steps.

Intuitively, the number of steps in a notification path is an upper bound on the number of events that need to take place before a future becomes assigned at the object, considering task evaluation a single event. Condition 1 in Definition 3.6.7 is the base case when f has already been instantiated. Condition 2 holds if a future has been resolved and a future message is in transit to o. Condition 3 holds if o is due to receive the return value of f from one of its pending tasks. Condition 4 holds if a call to o has been sent off from some object o'' with return future f. Condition 5 holds if a call to o' has been sent off from o with return future f; o is then on the notification path of f since the call message is guaranteed to be received at the callee's site (if at all) and the forwarding list there updated to include o. Condition 6 holds if o has been inserted into a forwarding list for f at a closer distance to a "source" of f. Condition 7, finally, holds if f is assigned to another future f', o'' is a "source" of f' for o, and o'' is due to receive the return value for f, which can then reach o through forwarding list additions in transitions using the rule FUT-SEND.

Example 3.6.8. Let cn be a configuration with two nodes and three objects, that, for some cn', can be written as

$$\mathsf{n}(u,t)\ \mathsf{n}(u',t')\ \mathsf{l}(u',q,u)\ \mathsf{o}(o_0,a_0,u,q_{in,0},q_{out,0})$$
$$\mathsf{o}(o_1,a_1,u,q_{in,1},q_{out,1})\ \mathsf{o}(o_2,a_2,u',q_{in,2},q_{out,2})\ cn'.$$

Suppose there are two futures f and f' such that cn assigns f' to f, and v to f'. Both futures are active for all three objects, and are unresolved at o_0 and o_1, but resolved at o_2. Specifically, let $\mathtt{hd}(q) = \mathsf{future}(o_0, f, f')$ and

$$\begin{aligned}
a_0(f) &= (\bot, o_1), & a_1(f) &= (\bot, \varepsilon), & a_2(f) &= (f', \varepsilon),\\
a_0(f') &= (\bot, \varepsilon), & a_1(f') &= (\bot, \varepsilon), & a_2(f') &= (v, o_0).
\end{aligned}$$

We then have that:

1. o_2 is on the notification path of f by o_2 in 0 steps (by Definition 3.6.7.1)
2. o_0 is on the notification path of f by o_0 in 1 step (by Definition 3.6.7.2)
3. o_1 is on the notification path of f by o_0 in 3 steps (by 2 and Definition 3.6.7.6)
4. o_2 is on the notification path of f' by o_2 in 0 steps (by Definition 3.6.7.1)
5. o_0 is on the notification path of f' by o_2 in 2 steps (by 4 and Definition 3.6.7.6)
6. o_1 is on the notification path of f' by o_2 in 8 steps (by 3, 5, and Definition 3.6.7.7)

Intuitively, the events that need to occur for o_1 to resolve f' to v are that:

1. o_0 receives the message $\mathsf{future}(o_0, f, f')$
2. o_0 sends the message $\mathsf{future}(o_1, f, f')$ and adds o_1 to its forwarding list for f'
3. o_2 sends the message $\mathsf{future}(o_0, f', v)$
4. o_0 receives the message $\mathsf{future}(o_0, f', v)$
5. o_0 sends the message $\mathsf{future}(o_1, f', v)$
6. o_1 receives the message $\mathsf{future}(o_1, f', v)$

Note that this is two events fewer than in the upper bound derived through Definition 3.6.7.7 above by doubling the upper bound for resolving f at o_1 and adding to that the upper bound for resolving f' at o_0.

We now prove that if o is on the notification path of f, then in the next configuration o remains on the notification path of f, without increasing the number of steps.

Lemma 3.6.9. *Fix a configuration cn and an object $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$. If o is on the notification path of f by o' in n steps in the configuration cn and $cn \to cn'$, then o is on the notification path of f by o' in at most n steps in cn'.*

Proof. See Section 3.13. □

We can now finally state the conditions of type 2 well-formedness.

Definition 3.6.10 (Type 2 Well-Formedness)**.** A type 2 configuration cn is *type 2 well-formed* (WF2) if cn satisfies:

1. *OID Uniqueness*: If $\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1})$ and $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2})$ are distinct object container occurrences in cn, then $o_1 \neq o_2$.
2. *Task-Object Existence*: If $\mathsf{t}(o, l, s) \preceq cn$, then $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ for some a, u, q_{in}, and q_{out}.
3. *Object-Node Existence*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, then $\mathsf{n}(u, t) \preceq cn$ for some t.
4. *Buffer Cleanliness*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ and $msg \preceq q_{in}$ or $msg \preceq q_{out}$, then msg is object bound. Additionally, if $msg \preceq q_{in}$, then $\mathtt{dst}(msg) = o$.
5. *Local Routing Consistency*: If $\mathsf{n}(u, t) \preceq cn$ and $\mathtt{nxt}(o, t) = u'$, then there is a link $\mathsf{l}(u, q, u') \preceq cn$.
6. *Call Uniqueness*: If $\mathsf{call}(o_1, o'_1, f_1, m_1, \overline{v}_1)$ and $\mathsf{call}(o_2, o'_2, f_2, m_2, \overline{v}_2)$ are distinct call message occurrences in queues in cn, then $f_1 \neq f_2$.
7. *Future Uniqueness*: If cn assigns both v_1 and v_2 to f, then $v_1 = v_2$.
8. *Single Writer*: If $\mathsf{t}(o_1, l_1, s_1)$ and $\mathsf{t}(o_2, l_2, s_2)$ are distinct task container occurrences in cn such that $l_1(\mathbf{ret}) = f_1$ and $l_2(\mathbf{ret}) = f_2$, then $f_1 \neq f_2$ and both f_1 and f_2 are unassigned in cn, and if $\mathsf{call}(o, o', f, m, \overline{v}) \preceq cn$, then $f \neq f_1$ and $f \neq f_2$.
9. *External OID*: $ext \notin OID(cn)$, and if $\mathsf{n}(u, t) \preceq cn$, then $ext \notin \mathtt{dom}(t)$.
10. *Future Liveness*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, and either f is active for o in cn, $a(f) \downarrow$, or cn assigns f to f' and o is on the notification path of f', then o is on the notification path of f.

Type 2 well-formedness is more complicated than its type 1 counterpart, mainly to account for the distributed way of handling futures (in particular, when applying a context). Buffer Cleanliness is needed to prevent the formation of contexts that are deadlocked because an in-queue or out-queue contains messages of the wrong type. Local Routing Consistency prevents the occurrence of routes through nonexistent links that could leave otherwise routable messages stuck. The rationale behind Single Writer is that mABS enforces a single-writer discipline on instantiated values of futures. Once a future has been instantiated through the evaluation of a return statement, the task is “garbage collected” in the rule RET-2. External OID ensures that messages to *ext* are only transported to reflexive links, where they remain. The last condition is the future propagation property discussed above. We use the designation Future Liveness not to indicate a guarantee that f will eventually be instantiated at an object, but to indicate that, if eventually f is instantiated somewhere, a notification path to the object exists along which the instantiation can be propagated.

Lemma 3.6.11 (WF2 Preservation)**.** *Let cn be a configuration. Then, the following holds:*

1. *If cn is a type 2 initial configuration, then cn is WF2.*
2. *If cn is WF2 and $cn \to cn'$, then cn' is WF2.*

Proof. See Section 3.13. □

An easy but important consequence of type 2 well-formedness is that assignments to futures cannot be updated with new values.

Proposition 3.6.12. *Suppose that cn is WF2 and $cn \to cn'$. If cn assigns v to f and cn' assigns v' to f, then $v = v'$.*

Proof. Since cn is WF2, if cn assigns v to f, there cannot be a task $\mathsf{t}(o, l, s) \preceq cn$ such that $l(\mathbf{ret}) = f$. But the only way of assigning $v' \neq v$ to f is through RET-2. Hence, the result follows. □

3.7 Type 2 Contextual Equivalence

We adapt the notion of contextual equivalence to the type 2 setting. The initial problem is to define the type 2 correlate of the observation predicate. Say a configuration cn has the observation, or *barb*, $obs = ext!m(\overline{v})$, if a corresponding call message $\mathsf{call}(ext, o, f, m, \overline{v})$, for some o and f, is located at the head of one of the self-loop link queues in cn. More precisely, the type 2 observability predicate $cn \downarrow obs$ holds just in case we have $cn = cn'\ \mathsf{l}(u, q, u)$ for some cn', and $\mathtt{hd}(q)$ is defined and equal to $\mathsf{call}(ext, o, f, m, \overline{v})$. Note that in the network-aware semantics, external call messages can always be shuffled on a reflexive link using MSG-DELAY-3, allowing specific calls to reach the head of the queue, to match observations in the reference semantics.

For type 2 context closure, a *context* is any configuration cn containing only object and task containers. Hence, contexts do not affect the underlying network graph. This definition is used, since, firstly, it is objects and tasks that induce computational behavior, and secondly, allowing contexts to augment the underlying graph by adding new nodes and links requires a much more complex account of network composition and well-formedness, left to future work.

With the observation predicate set up, the weak observation predicate is derived as in Section 3.5, and, as there, we define a *type 2 witness relation* as a relation that satisfies reduction closure, and barb preservation, with context closure defined as follows: if $cn_1 \mathcal{R} cn_2$, cn is a context, and $cn_1\ cn$ is WF2, then $cn_2\ cn$ is WF2, and $cn_1\ cn \mathcal{R} cn_2\ cn$. Thus:

Definition 3.7.1 (Type 2 Contextual Equivalence)**.** Let $cn_1 \simeq_2 cn_2$ whenever $cn_1 \mathcal{R} cn_2$ for a type 2 witness relation $\mathcal{R}$.

3.8 Normal Forms

We want to show that the type 1 behavior of an mABS program is preserved in the type 2 semantics. The key to the proof is a normal form lemma for mABS saying, roughly, that any well-formed type 2 configuration can be rewritten into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense stabilized, where all futures that are assigned a value somewhere are assigned a value everywhere the value might be needed, and where all objects have been moved to a single node. We perform this rewriting using two procedures. The first procedure stabilizes routing and empties link queues of everything except for external messages. The second procedure, which uses the first, empties object queues, propagates futures, and moves all objects to a single node.

3.8.1 Stabilization

In the scope of a configuration cn, we call a link $\mathsf{l}(u, q, u') \preceq cn$ *proper* whenever $u \neq u'$, and say that a message *msg* is *routable* whenever $\mathtt{dst}(msg) \in OID(cn)$, and *unroutable* otherwise.

Definition 3.8.1 (Stable Routing)**.** Let cn be a type 2 configuration. Say that cn has *stable routing*, if for all containers $\mathsf{n}(u, t)$, $\mathsf{o}(o, a, u', q_{in}, q_{out}) \preceq cn$, if $\mathtt{nxt}(o, t) = u''$, then there is a minimum length path from u to u' in $\mathtt{graph}(cn)$ which visits u''.

Definition 3.8.2 (External Link Messages)**.** Let cn be a type 2 configuration. We say that cn has *external link messages*, if $\mathsf{l}(u, q, u') \preceq cn$ and $msg \preceq q$ implies msg is object bound and unroutable.

To converge to a configuration with stable routing, the idea is to empty link queues as far as possible, and let nodes simultaneously exchange routing tables. This is accomplished using Algorithm 1 in Listing 3.2, where we hide uses of CTXT-1 to allow the transition rules to be applied to arbitrary containers. Write $\mathcal{A}_1(cn) \rightsquigarrow cn'$ if cn' is a possible result of applying Algorithm 1 to cn. The resulting configuration is almost unique, but not quite, since routing may stabilize in different ways.

Algorithm 1: Stabilize routing and deliver link messages

Input: A WF2 configuration *cn*
Output: A configuration with stable routing and external link messages, reachable from *cn*

```
repeat
  use OBJ-REG on each object not in transit ;
  use T-SEND on each proper link to broadcast routing tables
  from all nodes to their neighboring nodes ;
  repeat
    use T-RCV to dequeue one message on a link
  until T-RCV can no longer be used ;
  once for each link, if possible, use MSG-RCV, MSG-ROUTE,
  MSG-DELAY-1, or OBJ-RCV, or otherwise, use MSG-DELAY-3
  on self-loop links with external messages at the queue head
until routing has stabilized and there are only external link messages
```

Listing 3.2: Algorithm 1

Proposition 3.8.3. *Algorithm 1 terminates.*

Proof. See Section 3.13. □

We make the notion of stabilization precise using some auxiliary functions:

- $\mathsf{t}(cn)$ is the multiset of all tasks in cn.
- $\mathsf{o}_1(cn)$ is the object multiset where, if $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, there is a corresponding object $\mathsf{o}(o, a, u', q'_{in}, q_{out})$, such that the NID u' has been adjusted to that of the receiving node if the object was in transit from u to u' in cn, or $u' = u$ otherwise, and additionally, all messages in link queues in cn such that $\mathtt{dst}(msg) = o$ have been enqueued in some fixed order in q_{in} to produce q'_{in}.
- $\mathsf{m}_1(cn)$ is the multiset of both external and routable messages in cn.

Define the relation $\cong_1$ (different from $\simeq_1$) to hold between multisets of object containers when there is a one-to-one mapping where containers only possibly differ in how in-queue messages are ordered.

Definition 3.8.4 (Stable Form)**.** A WF2 configuration cn is in *stable form*, if

1. cn has stable routing,
2. $\mathsf{o}(cn) \cong_1 \mathsf{o}_1(cn)$, and
3. $\mathsf{m}(cn) = \mathsf{m}_1(cn)$.

Proposition 3.8.5. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then*

1. *$cn \to^* cn'$,*
2. *cn' is in stable form,*
3. *$\mathtt{graph}(cn) = \mathtt{graph}(cn')$,*
4. *$\mathsf{t}(cn) = \mathsf{t}(cn')$,*
5. *$\mathsf{o}_1(cn) \cong_1 \mathsf{o}(cn')$, and*
6. *$\mathsf{m}_1(cn) = \mathsf{m}(cn')$.*

Proof. Properties 1 and 3 are immediate. Property 2 can be read out of the termination proof. For the remaining three properties, observe first that t, o_1, and m_1 are all invariant under the transitions used in Algorithm 1. The equations then follow by noting that only externally-addressed messages (and so no object closures or routing tables) are in transit in links in cn'. □

Proposition 3.8.5 makes precise the "almost unique" property alluded to above. The properties used in this proposition inspire a notion of equivalence "up to stabilization", defined below.

Definition 3.8.6 ($\equiv_1$)**.** Define a binary relation $\mathcal{R}_1$ on type 2 configurations such that $cn_1 \; \mathcal{R}_1 \; cn_2$ whenever we have

1. $\mathtt{graph}(cn_1) = \mathtt{graph}(cn_2)$,
2. $\mathsf{t}(cn_1) = \mathsf{t}(cn_2)$,
3. $\mathsf{o}(cn_1) \cong_1 \mathsf{o}(cn_2)$, and
4. $\mathsf{m}(cn_1) = \mathsf{m}(cn_2)$.

We say that $cn_1 \equiv_1 cn_2$ if cn_1 and cn_2 are WF2, and there exists configurations cn_1' and cn_2' such that

$$\mathcal{A}_1(cn_1) \rightsquigarrow cn_1' \; \mathcal{R}_1 \; cn_2' \leftsquigarrow \mathcal{A}_1(cn_2).$$

Corollary 3.8.7. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$ then $cn \equiv_1 cn'$.*

Proof. We have $\mathcal{A}_1(cn) \rightsquigarrow cn' \; \mathcal{R}_1 \; cn' \leftsquigarrow \mathcal{A}_1(cn')$. □

Lemma 3.8.8. *$\equiv_1$ is reduction closed.*

Proof. See Section 3.13. □

Lemma 3.8.9. *$\equiv_1$ is context closed.*

Proof. See Section 3.13. □

Proposition 3.8.10. $\equiv_1$ *is a type 2 witness relation.*

Proof. See Section 3.13. □

Corollary 3.8.11. *If* $\mathcal{A}_1(cn) \rightsquigarrow cn'$, *then* $cn \simeq_2 cn'$.

Proof. By Corollary 3.8.7 and Proposition 3.8.10. □

Example 3.8.12. Figure 3.6 illustrates how stabilization works when run on a configuration with three objects on a network of four nodes with unstable routing. Figure 3.6(a) shows the initial configuration cn, and Figure 3.6(b) shows the configuration cn' that results after running Algorithm 1. In cn, an object container with identifier o_1, obj_1, is in transit from u_1 to u_0, and only the routing table at u_1 takes the new location of o_1 into account.

In cn', routing tables have stabilized due to repeated exchange of routing tables messages and link queues are empty, since there are no external messages. The future f' is still unresolved at o_0, since the future message has only been delivered to the in-queue of o_0 and not yet processed. Similarly, the call message to o_1 has been delivered to the object's in-queue, but the corresponding task has yet to be spawned. Consequently, there is no indication that the future f is either resolved or unresolved at o_1. Assuming no external messages in cn, self-loop link queues are empty and cn' is in stable form, since it has stable routing, $\mathsf{o}(cn') = \mathsf{o}_1(cn')$, and $\mathsf{m}(cn') = \mathsf{m}_1(cn')$ by the fact that no routable messages are in transit. Because no tasks have been added or removed, $\mathsf{t}(cn) = \mathsf{t}(cn')$. With all object-bound messages delivered to in-queues, $\mathsf{o}_1(cn) \cong_1 \mathsf{o}(cn')$. Finally, since there are no external or node-bound messages, $\mathsf{m}_1(cn) = \mathsf{m}(cn) = \mathsf{m}(cn')$.

3.8.2 Normalization

We turn to the second procedure, which empties object queues, propagates futures, and migrates all objects and their tasks to a single node. The procedure, Algorithm 2, is shown in Listing 3.3. Write $\mathcal{A}_2(cn) \rightsquigarrow cn'$ if cn' is a possible result of applying Algorithm 2 to cn. Initially, a node u is chosen towards which all objects will migrate during normalization. Normalization is then performed in cycles, with each cycle starting and ending in a configuration in stable form. In each cycle, one message is read from an object in- and out-queue. By well-formedness, object queues contain only call and future messages. Receptions of future messages may cause object environments to instantiate futures. This may cause new future instantiation messages to be enabled. Accordingly, those messages are generated and delivered to an object out-queue. Once this is done, objects not yet at u will be migrated towards u. Note that such migration does not require information from routing tables, and that network graph well-formedness guarantees that there is a migration path leading to u.

Proposition 3.8.13. *Algorithm 2 terminates.*

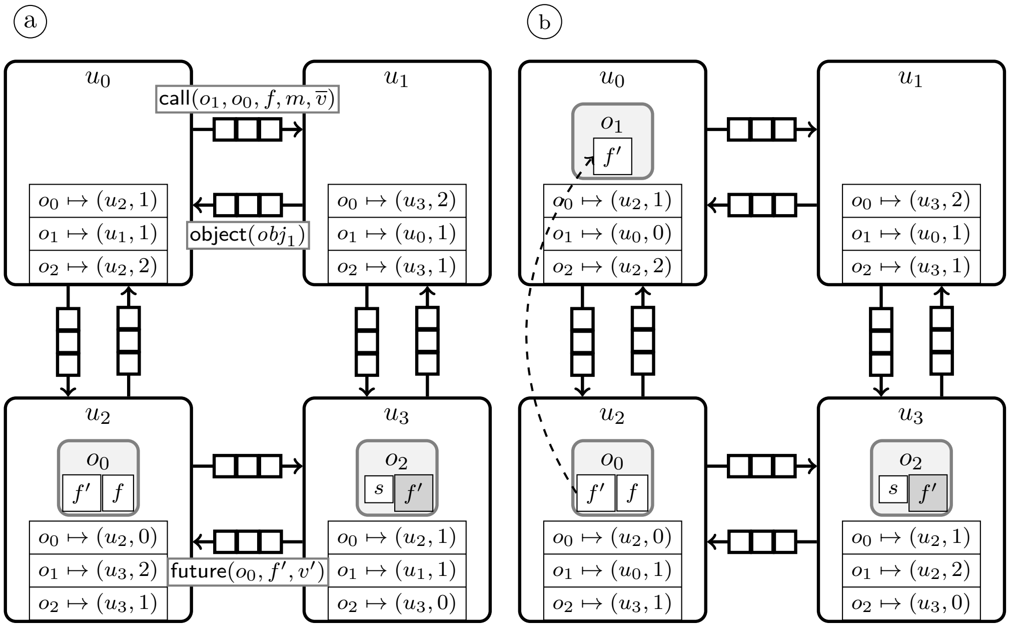


Figure 3.6: Configurations before and after running Algorithm 1

Proof. See Section 3.13. □

As for stabilization, we define some auxiliary functions:

- $\mathsf{t}_2(cn)$ is the multiset of task containers $tsk = \mathsf{t}(o, l, s)$ such that we either have $tsk \preceq cn$, or there is a routable message $\mathsf{call}(o, o', f, m, \overline{v})$ in transit in cn, such that $l = \mathtt{locals}(o, f, m, \overline{v})$ and $s = \mathtt{body}(o, m)$.
- $\mathsf{m}_2(cn)$ is the multiset of external messages in cn.
- $\mathsf{o}_2(cn)$ is the multiset of object containers $\mathsf{o}(o, a, u', \varepsilon, \varepsilon)$ such that $u' = u$, for which all of the following holds:
  - There is an object container $\mathsf{o}(o, a', u'', q_{in}, q_{out}) \preceq cn$ such that $a(\mathbf{self}) = a'(\mathbf{self})$ and for all variables x, $a(x) = a'(x)$.
  - $a(f) = (v, \varepsilon)$, if cn assigns v to f and o is on the notification path of f in cn.
  - $a(f) \uparrow$, if f is unassigned in cn, $a'(f) \uparrow$, and there is no f' such that cn assigns f to f' with o on the notification path of f'.
  - $a(f) = (\bot, \overline{o})$, if f is unassigned in cn, and either $a'(f) \downarrow$ or there is an f' such that cn assigns f to f' and, additionally, o is on the notification path

Algorithm 2: Normalization

Input: A WF2 configuration *cn*
Output: A configuration in normal form, reachable from *cn*

```
fix a NID $u$ for a node in $cn$ ;
run Algorithm 1 ;
repeat
  while some object queue is nonempty
    use MSG-SEND, MSG-DELAY-2, CALL-RCV-2, or FUT-RCV
    to dequeue one message from each nonempty object queue ;
    repeat
      use FUT-SEND to send future instantiation messages
    until FUT-SEND can no longer be used
  end ;
  while an object $o$ exists not located at $u$
    use OBJ-SEND to send $o$ towards $u$
  end ;
  run Algorithm 1
until all objects are located at $u$, all object queues are empty and there are only
external link messages
```

Listing 3.3: Algorithm 2

of f', with $\overline{o}$ defined as follows. Let $f_1, \ldots, f_n$ be all future assignments such that, for all f_i, o is on the notification path of f_i, and either *cn* assigns f_i to f, or there is a sequence of assignments of futures starting from f_i leading to f in *cn*. Let $\overline{o}_i$ be the forwarding list such that $\overline{o}_i = \pi_2(a'(f_i))$ if $a'(f_i) \downarrow$, and $\overline{o}_i = \varepsilon$ if $a'(f_i) \downarrow$. Let $\overline{o}'$ be the forwarding list such that $\overline{o}' = \pi_2(a'(f))$ if $a'(f) \downarrow$, and $\overline{o}' = \varepsilon$ if $a'(f) \uparrow$. Then, if $\mathsf{call}(o, o', f, m, \overline{v}) \preceq cn$ for some o', m, and $\overline{v}$, set $\overline{o} = o' \, \overline{o}_1 \ldots \overline{o}_n \, \overline{o}'$; set $\overline{o} = \overline{o}_1 \ldots \overline{o}_n \, \overline{o}'$ otherwise.

The somewhat involved definition when f is unassigned in $\mathsf{o}_2(cn)$ is due to the possibility of a chain of assignments $f_1 \mapsto \ldots \mapsto f_n \mapsto f$ in *cn*. Then, in the semantics, if the rule FUT-SEND is applied repeatedly as in Algorithm 2, all OIDs that are in the forwarding list for either of $f_1, \ldots, f_n$ at o will be added to the forwarding list for f at o.

We use these functions to describe the effects of running Algorithm 2 on a configuration to normalize it, as is made precise in the following definition and proposition.

Definition 3.8.14 (Normal Form)**.** A WF2 configuration *cn* is in *normal form*, if

1. *cn* has stable routing,

2. $\mathsf{t}(cn) = \mathsf{t}_2(cn)$,

3. $\mathsf{o}(cn) = \mathsf{o}_2(cn)$, and

4. $\mathsf{m}(cn) = \mathsf{m}_2(cn)$.

Proposition 3.8.15. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then*

1. *$cn \to^* cn'$,*

2. *cn' is in normal form,*

3. *$\mathtt{graph}(cn) = \mathtt{graph}(cn')$,*

4. *$\mathsf{t}_2(cn) = \mathsf{t}(cn')$,*

5. *$\mathsf{o}_2(cn) = \mathsf{o}(cn')$, and*

6. *$\mathsf{m}_2(cn) = \mathsf{m}(cn')$.*

Proof. See Section 3.13. □

We now give a notion of configuration equivalence up to normalization, which is key to our correctness argument.

Definition 3.8.16 ($\equiv_2$)**.** Define a binary relation $\mathcal{R}_2$ on type 2 configurations such that $cn_1 \; \mathcal{R}_2 \; cn_2$ whenever

1. $\mathtt{graph}(cn_1) = \mathtt{graph}(cn_2)$,

2. $\mathsf{t}(cn_1) = \mathsf{t}(cn_2)$,

3. $\mathsf{o}(cn_1) = \mathsf{o}(cn_2)$, and

4. $\mathsf{m}(cn_1) = \mathsf{m}(cn_2)$.

Let $cn_1 \equiv_2 cn_2$ if cn_1 and cn_2 are WF2 and there exists configurations cn_1' and cn_2' such that

$$\mathcal{A}_2(cn_1) \rightsquigarrow cn_1' \; \mathcal{R}_2 \; cn_2' \leftsquigarrow \mathcal{A}_2(cn_2).$$

Clearly, $\equiv_2$ relates more extended configurations than $\equiv_1$.

Corollary 3.8.17. $\equiv_1 \subseteq \equiv_2$.

Proof. If $cn_1 \equiv_1 cn_2$, the two configurations have the same task containers, and the same object bound messages. In addition, there is a one-to-one mapping between object containers where identifiers and object environments coincide. The result follows by noting that any remaining differences between the containers will disappear after running Algorithm 2. □

We also obtain that normalization respects normal form equivalence.

Corollary 3.8.18. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \equiv_2 cn'$.*

Proof. We have $\mathcal{A}_2(cn) \rightsquigarrow cn' \mathrel{\mathcal{R}_2} cn' \leftrightsquigarrow \mathcal{A}_2(cn')$. □

Lemma 3.8.19. *$\equiv_2$ is reduction closed.*

Proof. See Section 3.13. □

Lemma 3.8.20. *$\equiv_2$ is context closed.*

Proof. See Section 3.13. □

Proposition 3.8.21. *$\equiv_2$ is a type 2 witness relation.*

Proof. Similar to the proof of Proposition 3.8.10. □

Corollary 3.8.22. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \simeq_2 cn'$.*

Proof. By Corollary 3.8.18 and Proposition 3.8.21. □

Example 3.8.23. Figure 3.7 illustrates the effect of one cycle of normalization on a configuration. Figure 3.7(a) shows the initial configuration cn (the same as in Example 3.8.12), and Figure 3.7(b) shows the configuration cn' that results after running one cycle of Algorithm 2, assuming all object queues are empty in cn. Nodes migrate towards the node u_0, and s' is the statement for the task associated with the call to m for o_1, i.e., $s' = \mathtt{body}(o_1, m)$.

Note that for cn', in contrast to the corresponding configuration for stabilization where no unresolved futures become resolved and no tasks from call messages are spawned, f' is now resolved in both o_0 and o_1, and o_1 has a task with statement s' for the call message. The only change enacted on cn' in the next, final cycle of the algorithm is that o_2 is migrated to u_0 and the routing tables adjusted accordingly.

3.9 Correctness

In this section, we prove the correctness of the network-aware semantics by mapping a well-formed type 1 configuration $\mathtt{bind}\ \overline{z}.cn$ in standard form to a well-formed type 2 configuration $\mathtt{net}(cn)$ with an arbitrary, but well-formed, underlying network graph. We then prove that the two configurations are contextually equivalent.

3.9.1 The Representation Map

We first fix a well-formed graph represented as a configuration cn_{graph}, containing a node with distinguished NID u_0. Thus, cn_{graph} consists of nodes and links only, with each node u in cn_{graph} having the form $\mathsf{n}(u, t)$, and each link having the form $\mathsf{l}(u, \varepsilon, u')$. The routing tables t are defined later.

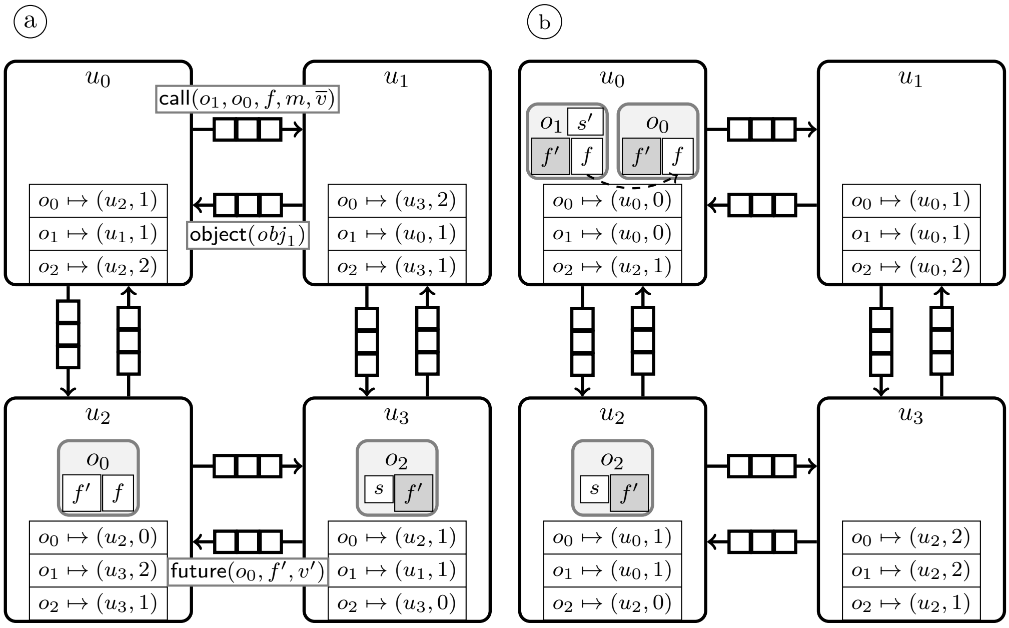


Figure 3.7: Configurations before and after running one cycle of Algorithm 2

Representing Names and Values We assume that names in the type 1 semantics are really symbolic, connected to concrete identifiers used in the type 2 semantics by means of an injective *name representation map* `rep`, taking internal names f, o in the type 1 semantics to names $\texttt{rep}(f)$, $\texttt{rep}(o)$ in the type 2 semantics. We extend the name representation map `rep` to arbitrary values and task environments in the obvious way:

- $\texttt{rep}(ext) = ext$
- $\texttt{rep}(p) = p$ for $p \in PVal$
- $\texttt{rep}(l)(x) = \texttt{rep}(l(x))$
- $\texttt{rep}(l)(\textbf{ret}) = \texttt{rep}(l(\textbf{ret}))$

Representing Object Environments One problem in extending `rep` to object environments is that such environments in the type 2 semantics must be defined partially in terms of the type 1 environments (for object variables), and partially in terms of the future containers available in the "root configuration", since the type 1 semantics uses future containers in place of forwarding lists. To this end, we first define an auxiliary operation $\texttt{oenvmap}(cn, \wp, \texttt{rep}) : FID \to Val_\perp$ on triples of type

1 configurations cn, pools $\wp$ of OID/FID constants, and name representation maps $\texttt{rep}$, as a function which gathers together assignments to futures as determined by the future containers in cn, as follows:

- $\texttt{oenvmap}(0, \wp, \texttt{rep})(f) = \bot$
- $\texttt{oenvmap}(tsk, \wp, \texttt{rep})(f) = \bot$
- $\texttt{oenvmap}(obj, \wp, \texttt{rep})(f) = \bot$
- $\texttt{oenvmap}(call, \wp, \texttt{rep})(f) = \bot$
- $\texttt{oenvmap}(\mathsf{f}(f, v_\bot), \wp, \texttt{rep})(f') =$ if $\texttt{rep}(f) = f'$ then $\texttt{rep}(v_\bot)$ else $\bot$
- $\texttt{oenvmap}(\mathsf{bind}\ z.cn, \wp \cup \{z'\}, \texttt{rep})(f) = \texttt{oenvmap}(cn, \wp, \texttt{rep}[z'/z])(f)$
- $\texttt{oenvmap}(cn_1\ cn_2, \wp, \texttt{rep})(f) =$
  $\texttt{oenvmap}(cn_1, \wp, \texttt{rep})(f) \sqcup \texttt{oenvmap}(cn_2, \wp, \texttt{rep})(f)$

Fix now a root type 1 configuration cn_0 and a large enough pool $\wp_0$ of names (proportional to the size of cn_0, and computed to conform to our naming policy). Assume that $cn_0 = \mathsf{bind}\ \overline{z_0}.cn_0'$ where cn_0' does not have binders. Fix $g = \texttt{oenvmap}(cn_0, \wp_0, \bot)$ and cn_{graph} as above. We can now extend $\texttt{rep}$ to object environments as follows:

- $\pi_1(\texttt{rep}(a))(\mathbf{self}) = \texttt{rep}(\pi_1(a)(\mathbf{self}))$
- $\pi_1(\texttt{rep}(a))(x) = \texttt{rep}(\pi_1(a)(x))$
- $\pi_2(\texttt{rep}(a))(f) = \begin{cases} (v, \varepsilon) & \text{if } g(f) = v \\ (\bot, \mathit{OID}(cn_0)) & \text{otherwise.} \end{cases}$

With expressions unspecified, we additionally need to assume that the representation map commutes with the expression semantics, i.e., that for all e, a, and l, it holds that

$$\texttt{rep}([\![e]\!]_{(a,l)}) = [\![e]\!]_{(\texttt{rep}(a),\texttt{rep}(l))}. \tag{3.1}$$

Proposition 3.9.1. *Let cn_0 be a type 1 well-formed root configuration in standard form, and $\wp_0$ be a pool as above. Then, $\texttt{rep}(a)(f) = (v, \varepsilon)$ if and only if $\mathsf{f}(f, v) \preceq cn_0$.*

Proof. See Section 3.13. □

Representing Call Containers Another complication is that we need to represent type 1 call containers as messages in the type 2 semantics. Compared to type 1 call containers, type 2 call messages additionally contain the OID of the caller, which is added to the future forwarding list when the message is received. Since future forwarding lists are already extended maximally, it is possible to use the callee's OID as the caller OID when mapping containers to messages without affecting behavior. In addition, these message-converted containers must delivered to a message queue. This is done by the operation `send`, which puts call messages in the self-loop queue of u_0, where

$$\texttt{send}(\mathsf{call}(o, o', f, m, \overline{v}), \mathsf{l}(u_0, q, u_0)\ cn) = \mathsf{l}(u_0, \mathsf{enq}(\mathsf{call}(o, o', f, m, \overline{v})), q), u_0)\ cn.$$

Given a name representation map `rep`, we now define the representation of a type 1 configuration as a transformer on type 2 configurations with the mapping `net`, as follows:

- $\texttt{net}(cn_1\ cn_2, \texttt{rep}) = \texttt{net}(cn_1, \texttt{rep}) \circ \texttt{net}(cn_2, \texttt{rep})$
- $\texttt{net}(0, \texttt{rep})(cn) = \texttt{net}(\mathsf{f}(f, v_\bot), \texttt{rep})(cn) = cn$
- $\texttt{net}(\mathsf{t}(o, l, s), \texttt{rep})(cn) = \mathsf{t}(\texttt{rep}(o), \texttt{rep}(l), s)\ cn$
- $\texttt{net}(\mathsf{o}(o, a), \texttt{rep})(cn) = \mathsf{o}(\texttt{rep}(o), \texttt{rep}(a), u_0, \varepsilon, \varepsilon)\ cn$
- $\texttt{net}(\mathsf{c}(o, f, m, \overline{v}), \texttt{rep})(cn) = \texttt{send}(\mathsf{call}(\texttt{rep}(o), \texttt{rep}(o), \texttt{rep}(f), m, \texttt{rep}(\overline{v})), cn)$

Defining Routing Tables The only detail remaining to be addressed from above concerns the routing tables. For the node with NID u_0, the initial routing table, t_0, needs to have all object identifiers in $\mathit{OID}(cn_0)$ registered, i.e.,

$$t_0 = \texttt{reg}(\texttt{rep}(o_1), u_0, \texttt{reg}(\cdots, \texttt{reg}(\texttt{rep}(o_n), u_0, \bot) \cdots)).$$

For nodes $\mathsf{n}(u, t)$ where $u \neq u_0$, we let t be determined by some stable routing, via Algorithm 1.

Defining the Map We can now finally define the representation map.

Definition 3.9.2 (Representation Map `net`)**.** Let a network configuration cn_{graph} and a name representation map `rep` be given for a WF1 configuration $\mathsf{bind}\ \overline{z}.cn$ in standard form. Then, the *type 2 representation* of $\mathsf{bind}\ \overline{z}.cn$ is $\texttt{net}(cn) = \texttt{net}(cn, \texttt{rep})(cn_{graph})$.

The most basic property that we expect to hold about the representation map is that it produces type 2 well-formed configuration given well-formed type 1 configurations.

Proposition 3.9.3. *If* $\mathsf{bind}\ \overline{z}.cn$ *is a WF1 configuration in standard form, then* $\texttt{net}(cn)$ *is WF2.*

Proof. See Section 3.13. □

3.9.2 Contextual Equivalence under Extension

Note that in the representation map, future-value mappings and forwarding lists are overapproximated when compared to the type 2 semantics, where future instantiations are only forwarded to and instantiated at objects that are on the notification path of the associated futures. Although not strictly necessary for proving correctness, we need to ensure that observable behavior does not change for well-formed configurations when future maps are extended in this way. To do this, we make precise the notion of extension for objects, and ultimately prove that configurations are contextually equivalent to their extended counterparts. Thus, a configuration produced by the mapping `net` captures the behavior of a corresponding minimally extended type 2 configuration produced from an initial configuration.

The crux of retaining behavior when extending the future map of an object o is to either *preserve* or *resolve* obligations to forward the value of a future f to some other object o'. In the simple case when the value of f is not another future, the obligation is considered resolved if o' has assigned f to the value. For the more complicated case when the value of f is another future f', we must additionally make sure that o' is due to receive the value of f' (and so on, in a chain of assignments). In addition, extended future maps must not introduce new forwardings for OIDs that are unroutable, which requires lifting the definition to the configuration level.

Definition 3.9.4 (Forwarding Resolved)**.** We say that o' is *forwarding resolved* for the future f in the configuration cn at the object o, if there is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, such that either $o' \in \pi_2(a(f))$, or one of the following holds:

1. cn assigns f to $v \in \mathit{PVal} \cup \mathit{OID}$ and there is a container $\mathsf{o}(o', a', u', q_{in}, q_{out}) \preceq cn$ such that $\pi_1(a'(f)) = v$.

2. cn assigns f to $f' \in \mathit{FID}$ and there is a container $\mathsf{o}(o', a', u', q_{in}, q_{out}) \preceq cn$ such that $\pi_1(a'(f)) = f'$ and, additionally, o' is forwarding resolved for f' in cn at o.

Definition 3.9.5 (Extended Object Container)**.** Let cn and cn' be configurations. An object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ *extends* an object container $\mathsf{o}(o', a', u', q'_{in}, q'_{out}) \preceq cn'$ if $o = o'$, $u = u'$, $q_{in} = q'_{in}$, $q_{out} = q'_{out}$, and

1. $a(\mathbf{self}) = a'(\mathbf{self})$, and for all variables x, $a(x) = a'(x)$,

2. if $\pi_1(a'(f)) = v$, then $\pi_1(a(f)) = v$,

3. if $o'' \in \pi_2(a'(f))$, then o'' is forwarding resolved for f in cn at o,

4. if $\pi_1(a(f)) = v$, then cn' assigns v to f, and if additionally $v \in \mathit{FID}$, then o is on the notification path of v in cn, and

5. if $o'' \in \pi_2(a(f))$, then either $o'' \in \pi_2(a'(f))$ or both $o'' \in \mathit{OID}(cn')$ and o is on the notification path of f in cn.

Definition 3.9.6 (Extended Configuration)**.** A configuration cn *extends* another configuration cn', written $cn \geqslant cn'$, if there is a one-to-one mapping of object containers $obj \preceq cn$ to object containers $obj' \preceq cn'$, such that obj extends obj', and additionally, $\texttt{graph}(cn) = \texttt{graph}(cn')$, $\mathsf{t}(cn) = \mathsf{t}(cn')$, and $\mathsf{m}(cn) = \mathsf{m}(cn')$.

Definition 3.9.7 (Notification Path Traversal)**.** Let cn be a configuration such that o is on the notification path of f by o' in n steps. Then, this notification path *traverses* o_{path} if

1. $n = 0$, $o' = o$, and $o_{path} = o$.
2. $n = 1$, $o' = o$, there is a future message $\mathsf{future}(o, f, v) \preceq cn$, and $o_{path} = o$.
3. $n = 1$, $o' = o$, there is a task $\mathsf{t}(o, l, s) \preceq cn$, and $o_{path} = o$.
4. $n = 2$, $o' = o$, there is a call message $\mathsf{call}(o, o'', f, m, \overline{v}) \preceq cn$, and $o_{path} = o$.
5. $n = 4$, and there is a call message $\mathsf{call}(o', o, f, m, \overline{v}) \preceq cn$, and $o_{path} = o$ or $o_{path} = o'$.
6. $n = n' + 2$, and there is an object $\mathsf{o}(o'', a', u', q_{in}, q_{out}) \preceq cn$ such that $o \in \pi_2(a'(f))$, and o'' is on the notification path of f by o' in n' steps, and either $o_{path} = o''$, or the notification path from o'' to o' traverses o_{path}.
7. $n = 2n' + n''$, and o is on the notification path of f' by o'' in n' steps, cn assigns f to f', and o'' is on the notification path of f by o' in n'' steps, and o_{path} either traverses the path of f' from o to o'', or the path of f from o'' to o'.

Proposition 3.9.8. *Let cn_1 and cn_2 be WF2 configurations such that $cn_1 \geqslant cn_2$. Then, if o is on the notification path of f by o_2 in n steps in cn_2, o is on the notification path of f by some o_1 in at most n steps in cn_1, such that the notification path from o to o_2 in cn_2 traverses o_1.*

Proof. See Section 3.13. □

Proposition 3.9.9. *If cn_1 and cn_2 are WF2 configurations such that $cn_1 \geqslant cn_2$ and there are cn_1' and cn_2' such that $\mathcal{A}_2(cn_1) \rightsquigarrow cn_1'$, and $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2'$, then $cn_1' \geqslant cn_2'$.*

Proof. See Section 3.13. □

We define a binary relation, $\cong_2$, that is a slight weakening of $\equiv_2$, and therefore includes $\equiv_2$, which relates WF2 configurations that, after running Algorithm 2, yield a pair of configurations such that one extends the other.

Definition 3.9.10 ($\cong_2$)**.** Let $cn_1 \cong_2 cn_2$ if cn_1 and cn_2 are WF2 configurations and there are cn_1' and cn_2' such that it is either the case that $\mathcal{A}_2(cn_1) \rightsquigarrow cn_1' \geqslant cn_2' \leftsquigarrow \mathcal{A}_2(cn_2)$, or $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2' \geqslant cn_1' \leftsquigarrow \mathcal{A}_2(cn_1)$.

Lemma 3.9.11. *$\cong_2$ is reduction closed.*

Proof. See Section 3.13. ☐

Lemma 3.9.12. *$\cong_2$ is context closed.*

Proof. See Section 3.13. ☐

Proposition 3.9.13. *$\cong_2$ is a type 2 witness relation.*

Proof. See Section 3.13. ☐

Lemma 3.9.14. *Suppose that cn' is WF2, and $cn \geqslant cn'$. Then, cn is WF2 as well, and $cn \simeq_2 cn'$.*

Proof. See Section 3.13. ☐

3.9.3 Correctness Result

We now obtain a key lemma which relates transitions in the network-oblivious and the network-aware semantics under the relation $\cong_2$.

Lemma 3.9.15. *Let* $\mathsf{bind}\ \overline{z}.cn$ *be a WF1 configuration in standard form. Then:*

1. *If* $\mathsf{bind}\ \overline{z}.cn \to \mathsf{bind}\ \overline{z}'.cn'$*, then for some cn'', we have* $\mathtt{net}(cn) \to^* cn''$ *and* $cn'' \cong_2 \mathtt{net}(cn')$*.*

2. *If* $\mathtt{net}(cn) \to cn''$*, then for some $\overline{z}'$ and cn', we have* $\mathsf{bind}\ \overline{z}.cn \to^* \mathsf{bind}\ \overline{z}'.cn'$ *and* $\mathtt{net}(cn') \cong_2 cn''$*.*

Proof. See Section 3.13. ☐

For both properties in Lemma 3.9.15, the argument is by case analysis on the possible rules applied in the assumed reduction step, using the aforementioned commutativity of the expression semantics with `rep` where necessary to produce a desired configuration.

Given our configuration mapping `net`, with a name representation map `rep`, we now conflate our notions of type 1 and type 2 witness relation into a notion that includes relations between WF1 and WF2 configurations, leading to a generalized contextual equivalence, $\simeq$. For such a conflated witness relation $\mathcal{R}$, reduction closure and barb preservation, as in Definition 3.5.1, is straightforward to define. The main problem lies in defining the notion of context closure, which requires applying a context configuration to two different configuration types. Applying a type 1 context cn to a type 2 configuration involves faithfully transforming elements of cn to the type 2 level, by introducing, e.g., locations to objects, and turning call containers into messages. Conversely, applying a type 2 context to a type 1 configuration involves removing locations and queues, turning messages into call containers, and introducing future containers.

More formally, suppose $\mathsf{bind}\ \overline{z}.cn_1\ \mathcal{R}\ cn_2$, with $\mathsf{bind}\ \overline{z}.cn_1$ WF1 and in standard form, and cn_2 WF2. Assume that, when we apply the type 1 context configuration cn to $\mathsf{bind}\ \overline{z}.cn_1$, we get the configuration $\mathsf{bind}\ \overline{z}'.cn_1\ cn'$ in standard form. We then apply the context to cn_2 by defining the result as $\mathtt{net}(cn', \mathtt{rep})(cn_2)$. Consequently, context closure requires that $\mathsf{bind}\ \overline{z}'.cn_1\ cn'\ \mathcal{R}\ \mathtt{net}(cn', \mathtt{rep})(cn_2)$ in this case. Conversely, suppose $cn_2\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{z}.cn_1$, again with cn_2 WF2, and $\mathsf{bind}\ \overline{z}.cn_1$ WF1 and in standard form. Straightforwardly, the result of applying the type 2 context configuration cn to cn_2 is the configuration $cn_2\ cn$. Define the configuration mapping $\mathtt{ten}$, which takes type 2 configurations to their type 1 counterparts by removing locations, queues, and future maps from objects, and adding message and future containers, with the help of the inverse of the name representation map, $\mathtt{rep}^{-1}$. The result of applying the context cn to $\mathsf{bind}\ \overline{z}.cn_1$ can then be defined as the standard-form configuration $\mathsf{bind}\ \overline{z}'.\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$, where all free names originating from cn in $\mathsf{bind}\ \overline{z}.\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$ have been bound. Consequently, converse context closure requires that $cn_2\ cn\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{z}'.\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$ in this case.

Using the established equivalence, $\simeq$, we can now finally state the correctness property.

Theorem 3.9.16 (Correctness of the Type 2 Semantics)**.** *For all well-formed type 1 configurations* $\mathsf{bind}\ \overline{z}.cn$ *in standard form,* $\mathsf{bind}\ \overline{z}.cn \simeq \mathtt{net}(cn)$*.*

Proof. See Section 3.13. □

The proof of Theorem 3.9.16 proceeds by showing, with the help of Lemma 3.9.15, that the relation

$$\mathcal{R} = \{(\mathsf{bind}\ \overline{z}.cn, cn') \mid \mathtt{net}(cn) \cong_2 cn'\}\ ,$$

where $\mathsf{bind}\ \overline{z}.cn$ is WF1 and in standard form and cn' is WF2, is a conflated witness relation. This is sufficient, since the identity relation is included in $\cong_2$.

3.10 Scheduling

The type 2 semantics is highly nondeterministic. The semantics says nothing about how frequently routing tables are to be exchanged, when messages should be passed between the different queues, when future messages are to be sent, or when, and to where, objects are to be transmitted. Resolving these choices involves making crucial tradeoffs between management overhead and performance. For instance, if routing tables are exchanged at a high frequency, routing can be assumed to be close to, or in, stable state. This ensures short end-to-end message routes for object-addressed messages, but at the expense of a large management and messaging overhead. Similarly, if objects quickly change location when a node is overloaded, the variance in load between nodes can be kept low and the load evenly balanced, at the cost of, e.g., task execution throughput. This raises the question of how to determine these parameters, something which we address in more detail in related work, described in Chapter 4.

Regardless of how, a real implementation needs to resolve these choices. This is tantamount to eliminating nondeterminism from the type 2 semantics, by introducing a scheduler that removes some transitions. The difficulty with scheduling is preemptive choice. For instance, in the type 2 semantics, a scheduler may force two messages that in the nondeterministic semantics are causally independent to be received in some given order. This phenomenon makes contextual equivalence and bisimulation-oriented methods in general inapplicable. One might hope to be able to devise schedulers that correspond to each other at both the type 1 and type 2 levels. We argue, however, that this is undesirable: a scheduler at the type 1 level may require global coordination across the entire network to be enforced at the type 2 level, e.g., to speed up message transmission across some links and slow them down correspondingly across others—without these links having any network proximity constraints whatsoever. This is exactly the kind of global synchronization overhead the type 2 semantics is designed to avoid. We therefore deliberately restrict attention to scheduling at the type 2 level. However, even in the presence of a scheduler at this level, we can still draw strong conclusions on faithfulness to the reference semantics, as we demonstrate below using a contextual simulation preorder in place of contextual equivalence. The idea is to view schedulers abstractly as predicates on type 2 configuration transition histories.

Definition 3.10.1 (Execution, Scheduler)**.** An *execution*, of either the type 1 or type 2 semantics, is a sequence of well-formed configurations $\rho = cn_1 \cdots cn_n$, such that, for $i : 1 \leq i < n$, it holds that $cn_i \rightarrow cn_{i+1}$. Let $\langle cn \rangle$ be the singleton execution consisting of only the configuration cn. A *scheduler* is a predicate $\mathcal{S}$ on type 2 executions, such that

- $\mathcal{S}(\langle cn \rangle)$ for all $\langle cn \rangle$, i.e., a scheduler kicks in only once an execution is started, and
- if $\mathcal{S}(cn_1 \cdots cn_n)$ and there exists a cn_{n+1} such that $cn_n \rightarrow cn_{n+1}$, then we have $\mathcal{S}(cn_1 \cdots cn_n\, cn_{n+1})$ for precisely one such cn_{n+1}.

That is, a scheduler is a device that determinizes type 2 executions. We define transition systems on executions, such that $\rho \rightarrow \rho'$ whenever $\rho = cn_1 \cdots cn_n$ and $\rho' = cn_1 \cdots cn_n\, cn_{n+1}$. The observation predicate $\rho \downarrow obs$, and application of a context configuration, is defined for executions similarly. A *scheduled transition system* is a transition system on type 2 executions, where, if we have $\rho \rightarrow \rho'$, $\mathcal{S}(\rho)$ and $\mathcal{S}(\rho')$ holds.

Let $\mathcal{R}$ be a relation on type 2 executions, scheduled by the scheduler $\mathcal{S}$, and unscheduled type 1 executions. Suppose $\mathcal{R}$ satisfies reduction closure, context closure and barb preservation, but that $\mathcal{R}^{-1}$ does not necessarily satisfy the converse properties. Assume the ($\mathcal{S}$-scheduled) type 2 execution ρ and the (unscheduled) type 1 execution ρ' are related by such an $\mathcal{R}$. We then say that ρ and ρ' are in the contextual simulation preorder $\lesssim$, written $\rho \lesssim \rho'$, and we obtain from Theorem 3.9.16 the following corollary.

Corollary 3.10.2. *For all well-formed type 1 configurations* $\mathsf{bind}\ \overline{z}.cn$ *in standard form,* $\langle \mathtt{net}(cn)\rangle \lesssim \langle \mathsf{bind}\ \overline{z}.cn\rangle$.

Proof. Suppose $\rho_1 \to \rho_1'$ for $\rho_1 = cn_1 \cdots cn_n$ and $\rho_1' = cn_1 \cdots cn_n cn_{n+1}$. We then have $cn_n \to cn_{n+1}$, and $\rho_1 \downarrow obs$ whenever $cn_n \downarrow obs$. Let $\rho_2 = cn_1' \cdots cn_m'$. When $cn_n = \mathtt{net}(cn)$ and $cn_m' = \mathsf{bind}\ \overline{z}.cn$, as in the present case, cn_n and cn_m' are by Theorem 3.9.16 related by some conflated witness relation $\mathcal{R}$. We use this relation when constructing the required relation on executions to qualify for inclusion in $\lesssim$, by straightforwardly transferring configuration properties from $\mathcal{R}$ to executions. □

Intuitively, Corollary 3.10.2 says that a scheduled execution in the network-aware semantics always maps to some specific (valid) execution in the network-oblivious semantics.

3.11 Discussion

A closely related precursor is Nomadic Pict [184], which follows earlier work on Pict [168], Fournet's distributed join-calculus [76], and JoCaml [42]. Besides the additional presence of futures in our language, we obtain, in comparison, a simpler and in our opinion more elegant correctness treatment, chiefly because our solution obviates the need for locking and consequently preemption, which has well-known detrimental consequences in a bisimulation-oriented setting.

Past correctness analyses for languages with futures have been carried out, e.g., by Caromel et al. [31] and Henrio et al. [89], but without an explicit treatment of distribution, communication, and routing. Henrio et al. prove correctness of future updates for an *eager home* strategy, where components (here, objects) must register themselves as recipients of future values. We consider the eager forward based strategy more appropriate to our setting than such a publish-subscribe strategy, since the former disperses the messaging load better among objects [90], and thus, in balanced allocations of objects to nodes, among nodes as well.

Standard ABS [103] and its extensions provide a comprehensive model of concurrent objects, related to the cobox model [183] and Creol [104], but without any concept of nodes, locations, or communication medium. Another difference is that the ABS unit of concurrency is an object group rather than a task, resulting in a more intuitive programming model without data races. A model with concurrent tasks as described here is still feasible to use by programmers, and can allow more efficient execution on multicore nodes [88]. Johnsen et al. [105] propose an extension of ABS with deployment components for explicit resource management. In contrast, their setting is not inherently decentralized, and the component model abstracts away from message distribution and routing, with the component topology being in effect a complete graph. However, if network nodes as described here are viewed as deployment components, the model becomes amenable to the location indpendent routing approach for message distribution, which may be useful for lower-level software systems modeling.

In the Klaim project [19], compilers are implemented and proven correct for several variants of the Klaim language, using the Linda tuple space communication model and a centralized name server to identify local tuple servers. The Oz kernel language [189] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [121] adds to this a notion of locality with separate failure and mobility semantics, but no real distribution or communication semantics is given; long distance communication is reduced to explicit manipulation of located agents, in the style of the Ambient Calculus [28]. The TKlaim (Topological Klaim) language [151] extends Klaim with network interconnections and primitives for manipulating them. Connections between nodes can be dynamically activated and deactivated, allowing a programmer to directly exploit the topology of the network.

Francalanza and Hennessy [78] introduce a distributed π-calculus variant, $D\pi$, with explicit nodes and links that can fail, requiring management of knowledge about inaccessible parts of the network (unreachable names). $D\pi$ is particularly suited for reasoning about software systems which discover and maintain information about the network topology. In comparison, mABS programs are oblivious to the network, but the semantics paves the way for a (separate) runtime system that can make decisions based on observed network conditions. Montanari and Sammartino [145] define a proper extension of π-calculus, called NCPi, which introduces nodes and links through two different types of names: sites and connectors. This allows creation and passing of connections between processes. In addition, processes in NCPi do not have a node location as do mABS tasks; instead they access the network through one or more nodes.

In Chapter 4, we study the use of the model presented here modified in detail for the ABS language core [103] to investigate decentralized runtime adaptability for objects, with promising results. The resulting language, defined in Appendix A and Appendix B, is more practical than mABS in that expressions and expression evaluation are fully defined, and it includes a type system which guarantees that well-typed programs have safety properties that go beyond mABS configuration well-formedness. Our adaptability-oriented network-aware semantics differs from the type 2 semantics in that network and object configurations are separate, but synchronize on complementary labeled transitions to transfer data between a node and an object. This leads to a clearer distinction between the network-layer and the object-layer than in the present work, similar to the distinction between meta-actors and actors in the architecture of Mechitov et al. [139].

On the one hand, our approach reduces complexity by doing away with some conventional parts of the network stack, i.e., changing the boundaries between layers. This approach has found success in other domains, such as data storage systems, where it has been referred to as *telescoping* a stack of layers [24]. Besides simplicity, rethinking layers can allow for significant performance gains for applications through reduced overhead, as demonstrated by Marinos et al. [135] with a custom network stack tailored for use by web servers. On the other hand, throwing out OSI network layers 3 and above may be an excessive price to pay, and it may turn out to be infeasible to amend current IP schemes in the direction proposed here. However, the architecture of the future Internet is currently very much in flux. It is possible today to build large scale non-IP networks with only layer 2 connectivity, sufficient to bootstrap a location independent scheme such as ours. The simplicity of formal reasoning when using our approach in comparison to the task of formally verifying, e.g., IP and TCP [20], suggests that currently ongoing work on verification of low-level software, along the lines of seL4 [109], can be extended to include fully networked operating systems and hypervisors.

In the Cloud Computing paradigm, a pool of network-interconnected computing resources are shared between different applications. Most proposed solutions to resource control for such pools, called clouds, are centralized, and can thus be expected to scale only to systems with in the order of thousands of nodes [102]. Our approach provides a building block for a scalable, formalized Platform as a Service (PaaS), that can be implemented correctly with few assumptions on network capabilities. The fully local nature of the reduction rules ensures that implementation code can correspond closely to the formalization, minimizing the risk of errors. An implementation of our semantics that realizes a PaaS cloud, with a deployment spanning several physical locations, can use a hybrid approach where some nodes communicate using direct physical links and others use TCP/IP. In a more general routing scheme, weights can be attached to links, instead of simply relying on the number of hops to approximate distance to an object. Physical links between nodes inside a datacenter can then be assigned, e.g., a fixed low weight, while links between "gateway" nodes in different datacenters, established using TCP over the Internet, can have higher weights, depending on their conventional routing distance.

The network-aware semantics assumes a fixed, static network throughout execution. Real-world networks are dynamic in at least two different ways. First, nodes can crash, then possibly recover, or deviate arbitrarily from prescribed behavior. Second, nodes can be added and shut down in a controlled way. In related work, described in Chapter 5, we consider the latter kind of dynamicity. We use a protocol reminiscent of a two-phase commit [187] to ensure objects can always be safely migrated away, and messages routed away, from nodes shutting down. The protocol in effect rules out simultaneous shutdown of neighboring nodes that have outstanding object-related messages between them, whence program-related state cannot disappear. Extended using suitable local criteria for connectivity, the protocol can also ensure networks remain connected after shutdowns, which is necessary for progress in program execution. To preserve object behavior in the face of node crash failures, some form of replication must be used for objects, tasks, and messages; the state checkpointing approach of Field and Varela [69] is one possibility. Handling of crashes can be formalized using failure detectors [36, 150].

3.12 Conclusion

The contribution of the paper has been to show that, using location independent routing, it is possible to devise novel and elegant network-based execution models for distributed object languages with fairly sophisticated features such as futures, and with attractive properties regarding correctness, performance, and scalability. Here, we focus on correctness, following the approach of earlier work on a simpler language without futures, as described in Chapter 2. As there, the main result relies critically on the inherent nondeterminism of the network-aware semantics.

Scalability is not fully resolved in the present work. We use a rather naïve distance vector routing scheme which has unit stretch but is not compact: routing tables may need to contain on the order of one entry per object identifier in the system. For large networks with many objects, other routing schemes are needed. Besides more scalable and robust routing, the first direction for future work is to examine richer language semantics, specifically with respect to more dynamicity. In ongoing work, we are studying power control: adding an explicit knob to the network-aware semantics for turning nodes on and off. Further down the line, it is of interest to handle both crash failures and Byzantine failures. The second, parallel, avenue is to study performance adaptation in more realistic settings. In our work on adaptability [159], our only management knob is object migration, and the management objective is to obtain good load balancing combined with good clustering properties. However, a real implementation will have many more management knobs such as buffer size, processor load, and power control.

Acknowledgements

We thank the anonymous reviewers of an earlier version of the paper for their comments and suggestions.

3.13 Proofs

Proposition 3.5.2. *The identity relation is a type 1 witness relation. $\simeq_1$ is a type 1 witness relation. If $\mathcal{R}$, $\mathcal{R}_1$, and $\mathcal{R}_2$ are type 1 witness relations then so are*

1. $\mathcal{R}^{-1}$,
2. $\mathcal{R}^*$, *and*
3. $\mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1$.

Proof. The identity relation is trivially reduction closed, context closed, and barb preserving, and is its own converse. To prove $\simeq_1$ is a type 1 witness relation, suppose $cn_1 \simeq_1 cn_2$. Then, $cn_1 \mathcal{R} cn_2$ for some type 1 witness relation. If $cn_1 \to cn_1'$, we have cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \mathcal{R} cn_2'$. But then $cn_1' \simeq cn_2'$, and we have shown reduction closure. For context closure, assume $cn_1\ cn$ is WF1; then $cn_1\ cn\ \mathcal{R}\ cn_2 cn$ and consequently $cn_1\ cn \simeq_1 cn_2\ cn$. For barb preservation, if $cn_1 \downarrow obs$, then $cn_2 \Downarrow obs$, since $\mathcal{R}$ is barb preserving. The converse arguments are completely symmetric.

For property 1, it suffices to note that $(\mathcal{R}^{-1})^{-1} = \mathcal{R}$. Reflexive, transitive closure in property 2 follows by a straightforward inductive argument. For property 3, if $cn_1 \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1 cn_2$ then $cn_1 \mathcal{R}_1 cn_{1,2} \mathcal{R}_2 cn_{2,1} \mathcal{R}_1 cn_2$ for some $cn_{1,2}$, $cn_{2,1}$. For reduction closure, assume $cn_1 \to cn_1'$. Then, $cn_{1,2} \to^* cn_{1,2}'$ and $cn_1' \mathcal{R}_1 cn_{1,2}'$. But then, $cn_{2,1} \to^* cn_{2,1}'$ and $cn_{1,2}' \mathcal{R}_2 cn_{2,1}'$. Consequently, $cn_2 \to^* cn_2'$ and $cn_{2,1}' \mathcal{R}_1 cn_2'$, whereby $cn_1' \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1 cn_2'$. For context closure, assume $cn_1 cn$ is WF1. Then, $cn_{1,2} cn$ is WF1 and $cn_1 cn\ \mathcal{R}\ cn_{1,2} cn$, and so on, yielding that $cn_2 cn$ is WF1 and $cn_1\ cn\ \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1\ cn_2\ cn$. For barb preservation, assume $cn_1 \downarrow obs$. Then, $cn_{1,2} \to^* cn_{1,2}' \downarrow obs$, allowing us to find $cn_{2,2}'$ such that $cn_{1,2}' \mathcal{R}_1 cn_{2,2}'$, whereby $cn' \to^* cn_{2,2}'' \downarrow obs$, and so on, yielding that $cn_2 \Downarrow obs$, as needed. Again, the converse arguments are symmetric. $\square$

Lemma 3.6.9. *Fix a configuration cn and an object $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$. If o is on the notification path of f by o' in n steps in the configuration cn and $cn \to cn'$, then o is on the notification path of f by o' in at most n steps in cn'.*

Proof. The proof is by induction on n. We follow the case analysis in Definition 3.6.7.

Case 1: We obtain that $\pi_1(a(f)) \downarrow$, and then, by Proposition 3.6.3.3, we find $\mathsf{o}(o, a', u', q_{in}', q_{out}') \preceq cn'$ such that $\pi_1(a'(f)) \downarrow$.

Case 2: Either $\mathsf{future}(o, f, v) \preceq cn'$ or we find $\mathsf{o}(o, a', u', q_{in}', q_{out}') \preceq cn'$ such that $\pi_1(a'(f)) \downarrow$.

Case 3: There are two options by Proposition 3.6.3.4: either we find a task $\mathsf{t}(o, l', s') \preceq cn'$ with $l'(\mathbf{ret}) = f$, or else we find $\mathsf{o}(o, a', u', q_{in}', q_{out}')$ such that $\pi_1(a'(l(\mathbf{ret}))) \downarrow$.

Case 4: Either $\mathsf{call}(o, o'', f, m, \overline{v}) \preceq cn'$ or we find $\mathsf{t}(o, l, s) \preceq cn'$ with $l(\mathbf{ret}) = f$.

Case 5: Either $\mathsf{call}(o', o, f, m, \overline{v}) \preceq cn'$, or we find $\mathsf{o}(o', a', u', q_{in}', q_{out}') \preceq cn'$ such that $\mathsf{t}(o', l', s') \preceq cn'$ with $l'(\mathbf{ret}) = f$, and $o \in \pi_2(a'(f))$.

Case 6: We find an object $obj'' = \mathsf{o}(o'', a'', u'', q_{in}'', q_{out}'') \preceq cn$ such that o'' is on the notification path of f by o' in $n-2$ steps in cn. By Proposition 3.6.3.3, we find the derivative $obj''' = \mathsf{o}(o'', a''', u'', q_{in}''', q_{out}''') \preceq cn'$ of obj'', and by the induction hypothesis o'' is also on the notification path of f by o' in cn', now in some $n'' \leq n-2$ steps. By inspection of the rules we see that either $\pi_2(a'''(f))$ is a suffix of $\pi_2(a''(f))$, or else there is a message $\mathsf{future}(o, f, \pi_1(a''(f))) \preceq cn'$. In either case we can conclude.

Case 7: We find an object $obj_1 = \mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn$ such that o_1 is on the notification path of f' by o'' in n' steps in cn, and an object $obj'' = \mathsf{o}(o'', a'', u'', q''_{in}, q''_{out}) \preceq cn$ such that o'' is on the notification path of f by o' in n'' steps, with $n = 2n' + n''$, where f' is assigned to f in cn. In the next step, by the induction hypothesis, o'' can be on the notification path of of f by o' in the same or fewer steps, and the length of the notification path of f' for o_1 does not increase as in the previous case. In either case we can conclude. □

Lemma 3.6.11. *Let cn be a configuration. Then, the following holds:*

1. *If cn is a type 2 initial configuration, then cn is WF2.*
2. *If cn is WF2 and $cn \to cn'$, then cn' is WF2.*

Proof. Property 1 follows straightforwardly from Definition 3.6.4 and Definition 3.6.10. For property 2, we consider each transition rule in turn.

OID Uniqueness: In all rules except NEW-2 there is a one-to-one correspondence between object container occurrences in cn and object container occurrences in cn'. This is sufficient to conclude. For NEW-2, it is sufficient to note that o' is a freshly generated OID.

Task-Object Existence, Object-Node Existence: The properties follow since neither nodes nor objects are ever removed. In the first instance, objects can only be created when the node is present, and in the second instance tasks can only be created when the object is present.

Buffer Cleanliness: We check that only object-bound messages enter in- and out-queues. This concerns the rules MSG-RCV, CALL-SEND-2, FUT-SEND, MSG-DELAY-1, and MSG-DELAY-2 only. The check is routine.

Local Routing Consistency: Easily proved by case analysis on rules.

Call Uniqueness: In all rules except CALL-SEND-2, call messages are the same in cn and cn', or present in cn and removed in cn'. This is sufficient to conclude. For CALL-SEND-2, it is sufficient to note that f is a freshly generated FID.

Future Uniqueness: We only need to consider rules which assign a non-$\bot$ value to futures. This happens in rules FUT-SEND, FUT-RCV and RET-2. The former two rules are immediate, and for RET-2 we use the assumption that cn satisfies Definition 3.6.10.5.

Single Writer: Again, we only need to consider the rules FUT-SEND, FUT-RCV and RET-2. Since for the former two rules, cn assigns v to f if and only if cn' does so, only RET-2 remains, which is immediate.

External OID: Assuming fresh identifiers for objects are never equal to ext, the check is routine.

Future Liveness: Let $\mathsf{o}(o, a', u', q'_{in}, q'_{out})$ be the derivative of $\mathsf{o}(o, a, u, q_{in}, q_{out})$ in cn, and assume that f is active for o for the configuration cn'. Either $a'(f)\downarrow$, or else a call message $\mathsf{call}(o', o, f', m, \overline{v})$ is in transit in cn' and f occurs in $\overline{v}$. We proceed by cases on the transition rule leading to cn'. Any rule that does not directly affect any of the conditions in Definition 3.6.6 or Definition 3.6.7 immediately allows to conclude that f is active for o also in cn. By the induction hypothesis, we can conclude that o is on the notification path of f in cn, and then o is on the notification path of f also in cn', since the only exception in Lemma 3.6.9 is when the environment of o is updated. For the remaining rules, there are the following cases to consider:

CALL-SEND-2: Assume first that o is the sending object. Either f is the newly introduced future in which case o is on the notification path of f according to Definition 3.6.7.5, since a call message is in transit from o to o' with return future f. If f is another future which is active for o in cn' then f is also active for o in cn. By the induction hypothesis, o is on the notification path of f in cn. Then o is also on the notification path of f in cn' by Lemma 3.6.9. On the other hand if o is not the sending object the case is immediately closed by the induction hypothesis, as the "pending" relation transfers from cn' to cn. We then apply the IH to conclude that o is on the notification path of f in cn, and then we use Lemma 3.6.9 to conclude that this also applies to cn'.

CALL-RCV-2: Assume first that o is the object receiving the call, and that f is the future of the call. Then o is on the notification path of f by Definition 3.6.7.4. Another option is that f is a future in $\overline{v}$ (referring to the transition rule in Figure 3.4). Then f is active for o in cn as well, by Definition 3.6.6. If f is some other future the case is completed by the IH as above.

FUT-SEND: Follows from the induction hypothesis and Lemma 3.6.9, as in the case for CALL-SEND-2.

FUT-RCV: If f is active for o for the configuration cn' then f is active for o also for cn, and f is not the received future. But then, the result follows by the IH and Lemma 3.6.9.

RET-2: If f is active for o for cn', then either f is not the return future, or f is the return future but has been received from some other object o' and is thus due to receive the value of f from o'. In both cases, we again complete by the induction hypothesis. □

Proposition 3.8.3. *Algorithm 1 terminates.*

Proof. In each iteration of the outermost loop of Algorithm 1, exactly one message is enqueued on each proper link, and at least one message is dequeued from all link queues. MSG-RCV, MSG-DELAY-1, and OBJ-RCV cause messages to leave the link queues, except for external messages, which are moved to the self-loop queues. If the link queues contain only routing table messages, the algorithm terminates in that iteration. If not, there must be object messages or routable call messages in some link queue. Since no new object messages are enqueued, there must some number of iterations n_0 after which all object messages have been received via OBJ-RCV and the associated object OIDs o registered on some node u so that $t(o) = (u, 0)$.

Let m_0 be the size of the largest link queue at the point which there are no object messages in transit. After $n_0 + m_0 + 1$ iterations, each node u has received at least one table update from each of its neighbors u', and the last table update applied to u has $t(o) = 0$. As a result, at point $n_0 + m_0 + 1$, each node u has $t(o) = (u', 1)$ whenever the u' is the host of o and the minimal length path from u to u' has length 1. The entry of the routing table of u for o will not change from that point onwards. We say that those entries are *stable.* Proceeding, let m_1 be the size of the largest link queue at point $n_0 + m_0 + 1$. After $n_0 + m_0 + 1 + m_1 + 1$ iterations, each routing table entry with length 2 (or less) will be stable. In the limit, each entry will be stable. It follows that Algorithm 1 must terminate, since, once routing has stabilized, rule MSG-ROUTE can only be applied a finite number of times before a routable message will be delivered. There is no chance of routable messages getting stuck in self-loop queues, since they are continuously shuffled using MSG-DELAY-3.

The only detail remaining to be checked is that a message can always be read from a link. Table and object messages can always be delivered, and call messages can also always be delivered, if nothing else to the self-loop link, in which case the routing table is not up-to-date or the message is external. This is the only case where MSG-DELAY-1 is used. This completes the argument. □

Lemma 3.8.8. $\equiv_1$ *is reduction closed.*

Proof. Suppose $cn_1 \equiv_1 cn_2$, where cn_1 and cn_2 are WF2. Assume $cn_1 \to cn_1'$; we need to find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \equiv_1 cn_2'$. We proceed by case analysis on the transition $cn_1 \to cn_1'$, eliding uses of CTXT-1.

For the cases T-SEND, T-RCV, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, OBJ-RCV, MSG-DELAY-3, and OBJ-REG, we take $cn_2' = cn_2$, since then, the stable form is unaffected, i.e., $cn_1 \equiv_1 cn_1'$, by Proposition 3.8.5.

The remaining cases include the rules for sequential control, MSG-SEND, MSG-DELAY-2, CALL-SEND-2, CALL-RCV-2, FUT-SEND, FUT-RCV, RET-2, GET-2, NEW-2, and OBJ-SEND. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition of the form

$$cn\ \mathtt{o}(o, a, u, q_{in}, q_{out})\ \mathtt{t}(o, l, x = e; s) \to cn\ \mathtt{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathtt{t}(o, l, s)$$

where $[\![e]\!]_{(a,l)} = v$ and $x \in \mathtt{dom}(a)$. Consider cn_2'' such that $\mathcal{A}_1(cn_2) \rightsquigarrow cn_2''$. By Proposition 3.8.5, there is a task $\mathtt{t}(o, l, x = e; s)$ and an object $\mathtt{o}(o, a, u, q_{in}', q_{out})$ in cn_2''. Hence, it is possible to perform a transition

$$cn'\ \mathtt{o}(o, a, u, q_{in}', q_{out})\ \mathtt{t}(o, l, x = e; s) \to cn'\ \mathtt{o}(o, a[v/x], u, q_{in}', q_{out})\ \mathtt{t}(o, l, s)$$

and we have

$$cn\ \mathtt{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathtt{t}(o, l, s) \equiv_1 cn'\ \mathtt{o}(o, a[v/x], u, q_{in}', q_{out})\ \mathtt{t}(o, l, s)$$

as needed, setting cn_2' to the right-hand side.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned} &cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathtt{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ &\quad\to\quad cn\ \mathsf{o}(o, a', u, q_{in}, \mathtt{enq}(msg, q_{out}))\ \mathtt{t}(o, l[f/x], s)\end{aligned}$$

where $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$, $f = \mathtt{newf}(u)$, $msg = \mathsf{call}(o', o, f, m, \overline{v})$, $o' = [\![e_1]\!]_{(a,l)}$, and $a' = \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a))$. Consider cn_2'' such that $\mathcal{A}_1(cn_2) \rightsquigarrow cn_2''$. By Proposition 3.8.5, there is a task $\mathtt{t}(o, l, x = e_1!m(\overline{e_2}); s)$ and an object $\mathsf{o}(o, a, u, q_{in}', q_{out})$ in cn_2''. Hence, it is possible to perform a transition

$$\begin{aligned} &cn'\ \mathsf{o}(o, a, u, q_{in}', q_{out})\ \mathtt{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ &\quad\to\quad cn'\ \mathsf{o}(o, a', u, q_{in}', \mathtt{enq}(msg, q_{out}))\ \mathtt{t}(o, l[f/x], s)\end{aligned}$$

and we have

$$\begin{aligned} &cn\ \mathsf{o}(o, a', u, q_{in}, \mathtt{enq}(msg, q_{out}))\ \mathtt{t}(o, l[f/x], s) \\ &\quad\equiv_1\quad cn'\ \mathsf{o}(o, a', u, q_{in}', \mathtt{enq}(msg, q_{out}))\ \mathtt{t}(o, l[f/x], s)\end{aligned}$$

setting cn_2' to the right-hand side. The remaining cases are proved in a similar straightforward way, by mimicking the transition after applying Algorithm 1. □

Lemma 3.8.9. $\equiv_1$ *is context closed.*

Proof. Assume $cn_1 \equiv_1 cn_2$ and $cn_1\ cn$ is WF2. We first show that $cn_2\ cn$ is WF2 as well.

OID Uniqueness: If $obj_1, obj_2 \preceq cn_2\ cn$ either $obj_1, obj_2 \preceq cn_2$, $obj_1, obj_2 \preceq cn$, or (wlog) $obj_1 \preceq cn_2$ and $obj_2 \preceq cn$. In either case, since $OID(cn_1) = OID(cn_2)$ by the definition of $\equiv_1$, the result follows.

Task-Object Existence: If $tsk \preceq cn_2\ cn$ either $tsk \preceq cn_2$ or $tsk \preceq cn$. In the former case, if $tsk = Tsk(o, l, s)$ then, since $cn_1 \equiv_1 cn_2$ and so cn_2 is WF2, we find $obj \preceq cn_2$ with $OID(obj) = \{o\}$. Otherwise, since $cn_1\ cn \equiv_1 cn_2\ cn$ we find $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_1\ cn$ and hence by definition of $\equiv_1$, $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_2\ cn$ as well.

Object-Node Existence: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_2\ cn$, then either the container is in cn_2, which is WF2 and thus has a node u, or it is in cn, which means node u is in cn_1, which has the same network as cn_2.

Buffer Cleanliness: If $obj \preceq cn_2\ cn$ either $obj \preceq cn_2$ or $obj \preceq cn$. In the former case we are done since cn_2 is WF2. In the latter case, we get $obj \preceq cn_1\ cn$ and $cn_1\ cn$ is WF2, which is sufficient.

Local Routing Consistency: This is immediate since cn contains only object and task containers.

Call Uniqueness: If $call \preceq cn_2$ and $call' \preceq cn$, and there is a future clash, there must be a clash between $call'$ and some message in cn_1. But this is ruled out since $cn_1\ cn$ is WF2.

Future Uniqueness: Assume cn_2 assigns v_1 to f and cn assigns v_2 to f and $v_1 \neq v_2$. Then, since $cn_1 \equiv_2 cn_2$, cn_1 assigns v_1 to f as well, violating WF2.

Single Writer: Assume $tsk_1, tsk_2 \preceq cn_2\ cn$, with associated future identifiers f_1 and f_2. Clearly, $f_1 \neq f_2$, and both are unassigned, or else WF2 would be violated for cn_2 or $cn_1\ cn$. Assume $call \preceq cn_2\ cn$ with f; then f is distinct from f_1 and f_2, or there would have been a clash in cn_2 or $cn_1\ cn$, ruled out by WF2.

External OID: cn_2 is WF2, so $ext \notin OID(cn_2)$. Also, $cn_1\ cn$ is WF2, so $ext \notin OID(cn)$. Hence $ext \notin OID(cn_2\ cn)$. Also, if t is a routing table in $cn_2\ cn$ it is a routing table in cn_2 and cn_2 is WF2. Then $ext \notin \mathtt{dom}(t)$, as required.

Future Liveness: Assume f is active for o in $cn_2\ cn$; then f is active for o in $cn_1\ cn$, and thus on the notification path of f there. Hence, it is also on the notification path of o in $cn_2\ cn$.

We next need to show that $cn_1\ cn\ \equiv_1 cn_2\ cn$. The WF2 property is immediate. Suppose $\mathcal{A}_1(cn_1\ cn) \rightsquigarrow cn_1'$ and $\mathcal{A}_1(cn_2\ cn) \rightsquigarrow cn_2'$. It suffices to prove $cn_1'\ \mathcal{R}_1\ cn_2'$. We check the requirements:

$\mathtt{graph}(cn_1') = \mathtt{graph}(cn_1\ cn) = \mathtt{graph}(cn_1) = \mathtt{graph}(cn_2) = \mathtt{graph}(cn_2\ cn) = \mathtt{graph}(cn_2')$.

$\mathsf{t}(cn_1') = \mathsf{t}(cn_1\ cn) = \mathsf{t}(cn_1) \cup \mathsf{t}(cn) = \mathsf{t}(cn_2) \cup \mathsf{t}(cn) = \mathsf{t}(cn_2\ cn) = \mathsf{t}(cn_2')$.

$\mathsf{o}(cn_1') \cong_1 \mathsf{o}_1(cn_1\ cn) = \mathsf{o}_1(cn_1) \cup \mathsf{o}_1(cn) \cong_1 \mathsf{o}_1(cn_2) \cup \mathsf{o}_1(cn) = \mathsf{o}_1(cn_1\ cn) \cong_1 \mathsf{o}(cn_2')$.

$\mathsf{m}(cn_1') = \mathsf{m}_1(cn_1\ cn) = \mathsf{m}_1(cn_1) \cup \mathsf{m}_1(cn) = \mathsf{m}_1(cn_2) \cup \mathsf{m}_1(cn) = \mathsf{m}_1(cn_2\ cn) = \mathsf{m}(cn_2')$. □

Proposition 3.8.10. $\equiv_1$ *is a type 2 witness relation.*

Proof. Reduction closure and its converse follows from Lemma 3.8.8, and context closure and its converse follows from Lemma 3.8.9. Hence, it suffices to show barb preservation in both directions. Suppose $cn_1 \equiv_1 cn_2$ and $cn_1 \downarrow obs$. Then, there is a call message msg with destination ext at the head of some self-loop queue in cn_1. After running Algorithm 1 on cn_2, we will have external link messages, with msg in some link queue. Using MSG-DELAY-1 and MSG-DELAY-3, msg can then be brought to the head of some self-loop queue. Hence, $cn_2 \Downarrow obs$. The proof of converse barb preservation is symmetric. □

Proposition 3.8.13. *Algorithm 2 terminates.*

Proof. Routing is stable after each run of Algorithm 1, and none of the rules applied in the outermost loop in the first outermost loop affect routing. Thus, one of MSG-SEND or MSG-DELAY-2 will be enabled whenever the output outqueue is nonempty, causing output queue size to decrease by one. By Buffer Cleanliness, one of CALL-RCV-2 or FUT-RCV will be applicable if the object in-queue is nonempty, decreasing in-queue size by one. Thus, when the inner while loop is reached, each nonempty in-queue has decreased in size by one, and each out-queue may have increased in size by one if the in-queue head position contains a delayed message.

Sending future messages may cause out-queues to increase in size. Each application of FUT-SEND causes a forwarding list to decrease in length by one. Thus, termination of the inner while loop is clear. We need to argue that the outer loop also terminates.

We first show that, eventually, no forwarding list is incremented. Only two rules can cause forwarding lists to increase in size, namely CALL-SEND-2 and CALL-RCV-2. Of these, CALL-SEND-2 is never used in either Algorithm 1 or Algorithm 2. Each application of CALL-RCV-2 consumes one call message, and none of the rules cause new call messages to be created. Thus, eventually, CALL-RCV-2 is never applied, and from that point onward forwarding lists are either emptied completely by the inner loop, or they remain untouched, since their corresponding future is undefined. Futures can become instantiated by FUT-RCV, but again, this can only happen a bounded number of times. Moreover, the only rule causing futures to be created is MSG-SEND, so the supply of futures to consider is fixed. Consequently, eventually, each future either remains uninstantiated forever, or else the corresponding forwarding list is empty. From that point onward, no FUT-SEND is enabled, and the innermost loop terminates trivially in all future iterations. In this situation, since MSG-SEND, CALL-RCV-2, and FUT-RCV all consume messages from a bounded resource (the set of messages in transit), if the outermost loop fails to terminate the only option is that, from some point onwards, only MSG-DELAY-2 is applied. From this point onward, since routing is stable, all messages will eventually be delivered.

Termination of the final loop is trivial. Observe that Algorithm 2 does not rely on routing to move the object towards u. For the algorithm, it is sufficient to establish that *some* good direction exists, and this is clearly the case as the network is stable and connected. □

Proposition 3.8.15. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then*

1. $cn \to^* cn'$,
2. *cn' is in normal form,*
3. $\mathtt{graph}(cn) = \mathtt{graph}(cn')$,
4. $\mathsf{t}_2(cn) = \mathsf{t}(cn')$,
5. $\mathsf{o}_2(cn) = \mathsf{o}(cn')$, *and*
6. $\mathsf{m}_2(cn) = \mathsf{m}(cn')$.

Proof. Property 1 is immediate. Property 3 follows from property 3 of Proposition 3.8.5.

For property 4, observe first that the function t_2 is invariant under transitions used in Algorithm 2. On termination of Algorithm 2, only external messages are in transit, and since no rule used in the algorithm modifies an existing task, property 4 follows.

For property 5, let $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}(cn')$. We show that it is then the case that $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}_2(cn)$. From the definition of Algorithm 2, $q_{in,2} = q_{out,2} = \varepsilon$. Also, $u_2 = u$. We know that there is an object container $\mathsf{o}(o, a', u'', q_{in}, q_{out}) \preceq cn$, since there is a one-to-one correspondence between object containers in pre- and poststate for each transition used in Algorithm 2. We also know that $a'(x) = a_2(x)$ for all x. Suppose finally that an object container $obj = \mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1})$ exists in cn with $a_1(f) = (v, \overline{o})$. Let $\mathsf{o}(o_1, a_1', u_1', q_{in,1}', q_{out,1}')$ be the derivative of obj in cn'. Then $\pi_1(a_1'(f)) = v$ as well, by Proposition 3.6.12. We know by Future Uniqueness that $a_2(f) = (v', \overline{o'})$ implies $v' = v$. It remains to show that $\pi_1(a_2(f)) \neq \bot$. Assume not. We have that o_2 is on the notification path of f in n steps for some nonnegative integer n. We proceed by induction on the notification path relation:

Case 1: $n = 0$, and $\pi_1(a_2(f)) = v \neq \bot$. This yields a contradiction.

Case 2: $n = 1$, and there is future message $\mathsf{future}(o_2, f, v') \preceq cn$. This yields a contradiction, since the only queued messages in cn' are external.

Case 3: $n = 1$, and there is a task $\mathsf{t}(o_2, l_2, s_2) \preceq cn'$ with $l_2(\mathbf{ret}) = f$. Then, $\pi_1(a_1'(f)) = \bot$ by well-formedness, which is a contradiction.

Case 4: $n = 2$, and there is a call message $\mathsf{call}(o_2, o'', f, m, \overline{v}) \preceq cn'$. This again contradicts the assumption of having only external messages.

Case 5: $n = 4$, and there is a call message $\mathsf{call}(o', o_2, f, m, \overline{v}) \preceq cn'$ such that $o' \in OID(cn')$. This again contradicts the assumption of having only external messages.

Case 6: $n = n' + 2$ and there is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn'$ such that $o_2 \in \pi_2(a(f))$, and o is on the notification path of f in n' steps. Then, if we have $\pi_1(a(f)) = v \neq \bot$, cn' is not in normal form, a contradiction; alternatively, we conclude by the IH.

Case 7: $n = 2n' + n''$ and there is an object container $\mathsf{o}(o'', a, u, q_{in}, q_{out}) \preceq cn'$ such that o is on the notification path of f' by o'' in n' steps, f' is assigned to f, and there is an object container $\mathsf{o}(o', a', u', q_{in}, q_{out}) \preceq cn'$ such that o' is on the notification path of f in n'' steps. By the IH, we have $\pi_2(a'(f)) = v \neq \bot$ and thus $\pi_2(a(f)) = v$. Since f will have been forwarded down to o_2, v will have been forwarded down the same chain.

We can thus conclude that $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}_2(cn)$. Conversely, assume that $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}_2(cn)$. Object o_2 has exactly one derivative in cn', by well-formedness. That object has empty queues, the same NID as in cn, preserves assignments to variables, and has $\pi_1(a_2(f))$ assigned to a non-$\bot$ value if and only if some object in cn' has so, by the above argument.

For 3.8.15.6, the property holds as it does so already for Algorithm 1.

We finally need to prove property 2. Property 3.8.14.1 is trivial, as each run of Algorithm 2 ends with a run of Algorithm 1, and Algorithm 1 ensures that cn' has stable routing. Property 3.8.14.2 holds since Algorithm 1 ensures almost empty link queues, and since on termination, Algorithm 2 ensures empty object queues. For 3.8.14.3, if $obj = \mathsf{o}(o, a, u', \varepsilon, \varepsilon)$ satisfies the properties defining o_2 above then, referring to those conditions, $u' = u'' = u$, $a' = a$, $q_{in} = \varepsilon = q_{out}$, and $obj \preceq cn'$, as needed to be shown. For 3.8.14.4, the result follows since only external messages are in transit in cn'. ☐

Lemma 3.8.19. $\equiv_2$ *is reduction closed.*

Proof. Suppose $cn_1 \equiv_2 cn_2$, where cn_1 and cn_2 are WF2. Assume $cn_1 \to cn_1'$; we need to find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \equiv_2 cn_2'$. We proceed by case analysis on the transition $cn_1 \to cn_1'$, eliding uses of CTXT-1.

For the cases T-SEND, T-RCV, MSG-SEND, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, MSG-DELAY-2, MSG-DELAY-3, CALL-RCV-2, FUT-SEND, FUT-RCV, OBJ-REG, OBJ-SEND, and OBJ-RCV, we take $cn_2' = cn_2$, since then, the normal form is unaffected, i.e., $cn_1 \equiv_2 cn_1'$, by Proposition 3.8.15.

The remaining cases include the rules for sequential control, CALL-SEND-2, RET-2, GET-2, and NEW-2. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition of the form

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$$

where $[\![e]\!]_{(a,l)} = v$ and $x \in \mathtt{dom}(a)$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 3.8.15, there is a task $\mathsf{t}(o, l, x = e; s)$ and an object $\mathsf{o}(o, a', u', \varepsilon, \varepsilon)$ in cn_2''. Hence, it is possible to perform a transition

$$cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e; s) \to cn'\ \mathsf{o}(o, a'[v/x], u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, s)$$

and we have

$$cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s) \equiv_2 cn'\ \mathsf{o}(o, a'[v/x], u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, s)$$

as needed, setting cn_2' to the right-hand side.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned} & cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ \to \quad & cn\ \mathsf{o}(o, \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a)), u, q_{in}, \mathtt{enq}(msg, q_{out}))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

where $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$, $f = \mathtt{newf}(u)$, $msg = \mathsf{call}(o', o, f, m, \overline{v})$, and $o' = [\![e_1]\!]_{(a,l)}$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 3.8.15, there is a task $\mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s)$ and an object $\mathsf{o}(o, a', u', \varepsilon, \varepsilon)$ in cn_2'' with $[\![\overline{e_2}]\!]_{(a',l)} = \overline{v}$ and $[\![e_1]\!]_{(a',l)} = o'$. Hence, it is possible to perform a transition

$$\begin{aligned} & cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ \to \quad & cn'\ \mathsf{o}(o, \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a')), u', \varepsilon, \mathtt{enq}(msg, \varepsilon))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

and we have

$$\begin{aligned} &cn\ \mathtt{o}(o, \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a)), u, q_{in}, \mathtt{enq}(msg, q_{out}))\ \mathtt{t}(o, l[f/x], s) \\ \equiv_2 \quad &cn'\ \mathtt{o}(o, \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a')), u', \varepsilon, \mathtt{enq}(msg, \varepsilon))\ \mathtt{t}(o, l[f/x], s) \end{aligned}$$

as needed, setting cn_2' to the right-hand side.

RET-2: Consider a transition of the form

$$cn\ \mathtt{o}(o, a, u, q_{in}, q_{out})\ \mathtt{t}(o, l, \mathbf{return}\ e; s) \to cn\ \mathtt{o}(o, a[v/f], u, q_{in}, q_{out})$$

where $v = [\![e]\!]_{(a,l)}$ and $f = l(\mathbf{ret})$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 3.8.15, there is a task $\mathtt{t}(o, l, \mathbf{return}\ e; s)$ and an object $\mathtt{o}(o, a', u', \varepsilon, \varepsilon)$ in cn_2'' where $[\![e]\!]_{(a',l)} = v$. Hence, it is possible to perform a transition

$$cn'\ \mathtt{o}(o, a', u', \varepsilon, \varepsilon)\ \mathtt{t}(o, l, \mathbf{return}\ e; s) \to cn'\ \mathtt{o}(o, a'[v/f], u', \varepsilon, \varepsilon)$$

and we have

$$cn\ \mathtt{o}(o, a[v/f], u, q_{in}, q_{out}) \equiv_2 cn'\ \mathtt{o}(o, a'[v/f], u', \varepsilon, \varepsilon)$$

as needed, setting cn_2' to the right-hand side.

GET-2: Consider a transition of the form

$$cn\ \mathtt{o}(o, a, u, q_{in}, q_{out})\ \mathtt{t}(o, l, x = e.\mathbf{get}; s) \to cn\ \mathtt{o}(o, a, u, q_{in}, q_{out})\ \mathtt{t}(o, l[v/x], s)$$

where $[\![e]\!]_{(a,l)} = f$ and $\pi_1(a(f)) = v$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 3.8.15, there is a task $\mathtt{t}(o, l, x = e.\mathbf{get}; s)$ and an object $\mathtt{o}(o, a', u', \varepsilon, \varepsilon)$ in cn_2'' where $[\![e]\!]_{(a',l)} = f$ and $\pi_1(a'(f)) = v$. Hence, it is possible to perform a transition

$$cn'\ \mathtt{o}(o, a', u', \varepsilon, \varepsilon)\ \mathtt{t}(o, l, x = e.\mathbf{get}; s) \to cn'\ \mathtt{o}(o, a', u', \varepsilon, \varepsilon)\ \mathtt{t}(o, l[v/x], s)$$

and we have

$$cn\ \mathtt{o}(o, a, u, q_{in}, q_{out})\ \mathtt{t}(o, l[v/x], s) \equiv_2 cn'\ \mathtt{o}(o, a', u', \varepsilon, \varepsilon)\ \mathtt{t}(o, l[v/x], s)$$

as needed, setting cn_2' to the right-hand side.

NEW-2: Consider a transition of the form

$$\begin{aligned} &cn\ \mathtt{o}(o, a, u, q_{in}, q_{out})\ \mathtt{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \\ \to \quad &cn\ \mathtt{o}(o, a', u, q_{in}, q_{out})\ \mathtt{t}(o, l[o'/x], s)\ \mathtt{o}(o', a'', u, \varepsilon, \varepsilon) \end{aligned}$$

where $o' = \mathtt{newo}(u)$, $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \mathtt{fw}(\overline{v}, o', a)$, and $a'' = \mathtt{init}(\overline{v}, \mathtt{init}(C, \overline{v}, o'))$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 3.8.15, there is a task

$\mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s)$ and an object $\mathsf{o}(o, a_1, u', \varepsilon, \varepsilon)$ in cn_2'' where $[\![e]\!]_{(a_1,l)} = \overline{v}$ and $\pi_1(a_1(f)) = v$. Hence, it is possible to perform a transition

$$\begin{aligned} & cn'\ \mathsf{o}(o, a_1, u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \\ & \quad \to \quad cn'\ \mathsf{o}(o, a_1', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a'', u', \varepsilon, \varepsilon) \end{aligned}$$

and we have

$$\begin{aligned} & cn\ \mathsf{o}(o, a', u, q_{in}, q_{out})\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a'', u, \varepsilon, \varepsilon) \\ & \quad \equiv_2 \quad cn'\ \mathsf{o}(o, a_1', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a'', u', \varepsilon, \varepsilon) \end{aligned}$$

as needed, setting cn_2' to the right-hand side. □

Lemma 3.8.20. $\equiv_2$ *is context closed.*

Proof. The proof follows the proof of Lemma 3.8.9. Assume $cn_1 \equiv_2 cn_2$ and $cn_1\ cn$ is WF2. We first show that $cn_2\ cn$ is WF2 as well.

OID Uniqueness: If $obj_1, obj_2 \preceq cn_2\ cn$ either $obj_1, obj_2 \preceq cn_2$, $obj_1, obj_2 \preceq cn$, or (wlog) $obj_1 \preceq cn_2$ and $obj_2 \preceq cn$. In either case, since $OID(cn_1) = OID(cn_2)$ by the definition of $\equiv_2$, the result follows.

Task-Object Existence: If $tsk \preceq cn_2\ cn$ either $tsk \preceq cn_2$ or $tsk \preceq cn$. In the former case, if $tsk = \mathsf{t}(o, l, s)$ then, since $cn_1 \equiv_2 cn_2$ and so cn_2 is WF2, we find $obj \preceq cn_2$ with OID o. Otherwise, since $cn_1\ cn \equiv_2 cn_2\ cn$ we find $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_1\ cn$ and hence by definition of $\equiv_2$, $\mathsf{o}(o, a', u', q_{in}', q_{out}') \preceq cn_2\ cn$ as well.

Object-Node Existence: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_2\ cn$, then either the container is in cn_2, which is WF2 and thus has a node u, or it is in cn, which means node u is in cn_1, which has the same network as cn_2.

Buffer Cleanliness: If $obj \preceq cn_2\ cn$ either $obj \preceq cn_2$ or $obj \preceq cn$. In the former case we are done since cn_2 is WF2. In the latter case, we get $obj \preceq cn_1\ cn$ and $cn_1\ cn$ is WF2, which is sufficient.

Local Routing Consistency: This is immediate since cn contains only object and task containers.

Call Uniqueness: If $call \preceq cn_2$ and $call' \preceq cn$, and there is a future clash, there must be a clash between $call'$ and some message in cn_1 or a task in cn_1. But this is ruled out since $cn_1\ cn$ is WF2.

Future Uniqueness: Assume cn_2 assigns v_1 to f and cn assigns v_2 to f and $v_1 \neq v_2$. Then, since $cn_1 \equiv_2 cn_2$, cn_1 assigns v_1 to f as well, violating WF2.

Single Writer: Assume $tsk_1, tsk_2 \preceq cn_2\ cn$, with associated future identifiers f_1 and f_2. Clearly, $f_1 \neq f_2$, and both are unassigned, or else WF2 would be violated for cn_2 or $cn_1\ cn$. Assume $call \preceq cn_2\ cn$ with f; then f is distinct from f_1 and f_2, or there would have been a clash in cn_2 or $cn_1\ cn$, ruled out by WF2.

External OID: If $\mathit{ext} \in \mathit{OID}(cn_2\ cn)$, then $\mathit{ext} \in \mathit{OID}(cn)$, which is ruled out by WF2; the second requirement is immediate since cn_2 is WF2 and cn contains no nodes.

Future Liveness: Assume f is active for o in $cn_2\ cn$; then f is active for o in $cn_1\ cn$, and thus on the notification path of f there. Hence, it is also on the notification path of o in $cn_2\ cn$.

We next need to show that $cn_1\ cn\ \equiv_2\ cn_2\ cn$. The WF2 property is immediate. Suppose $\mathcal{A}_2(cn_1\ cn) \rightsquigarrow cn_1'$ and $\mathcal{A}_2(cn_2\ cn) \rightsquigarrow cn_2'$. It suffices to prove $cn_1'\ \mathcal{R}_2\ cn_2'$. We check the requirements:

$\mathtt{graph}(cn_1') = \mathtt{graph}(cn_1\ cn) = \mathtt{graph}(cn_1) = \mathtt{graph}(cn_2) = \mathtt{graph}(cn_2\ cn) = \mathtt{graph}(cn_2')$.

Since $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$, we have $\mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn)$. Hence, $\mathsf{t}(cn_1') = \mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn) = \mathsf{t}(cn_2')$.

Since $\mathsf{o}_2(cn_1) = \mathsf{o}_2(cn_2)$, we have $\mathsf{o}_2(cn_1\ cn) = \mathsf{o}_2(cn_2\ cn)$. Hence, $\mathsf{o}(cn_1') = \mathsf{o}_2(cn_1\ cn) = \mathsf{o}_2(cn_2\ cn) = \mathsf{o}(cn_2')$.

$\mathsf{m}(cn_1') = \mathsf{m}_2(cn_1\ cn) = \mathsf{m}_2(cn_1) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2\ cn) = \mathsf{m}(cn_2')$. □

Proposition 3.9.1. *Let cn_0 be a type 1 well-formed root configuration in standard form, and $\wp_0$ be a pool as above. Then,* $\mathtt{rep}(a)(f) = (v, \varepsilon)$ *if and only if* $\mathsf{f}(f, v) \preceq cn_0$.

Proof. By well-formedness, the future container, if it exists, is unique. Pick a name representation map $\mathtt{rep}$. Then, $\mathtt{oenvmap}(cn_0, \wp_0, \mathtt{rep})(f)$ is defined and equal to v if and only if $\mathsf{f}(f, v) \preceq cn_0$. This is easily seen by induction on the structure of cn_0. □

Proposition 3.9.3. *If* $\mathsf{bind}\ \overline{z}.cn$ *is a WF1 configuration in standard form, then* $\mathtt{net}(cn)$ *is WF2.*

Proof. We consider the WF2 conditions in turn. OID Uniqueness and Task-Object Existence follows from the respective WF1 conditions and from how the name representation map is defined. Object-Node Existence holds since all objects are placed on the node u_0, which exists by the definition of cn_{graph}. Buffer Cleanliness holds since all object containers in $\mathtt{net}(cn)$ have empty queues. Local Routing Consistency follows from how routing tables in cn_{graph} are defined. Call Uniqueness follows from the corresponding WF1 condition and from how the name representation map is defined. Future Uniqueness follows the definition of $\mathtt{oenvmap}$, how object environments are represented, and from Proposition 3.9.1. Single Writer follows the corresponding WF1 condition and from how the name mapping representation map is defined. External OID follows from the corresponding WF1 condition and from how routing tables are defined.

Consider finally Future Liveness. Assume $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq \mathtt{net}(cn)$. Note that by the way we define future mappings in object environments, we have that, whenever f is active for o in $\texttt{net}(cn)$, $a(f)\downarrow$ holds. Suppose therefore that $a(f)\downarrow$. Then, there must be some future container for the FID corresponding to f in $\mathsf{bind}\ \overline{z}.cn$ with lifted value $v_\perp$. If $v_\perp \neq \perp$, then $a(f) = (v, \varepsilon)$ for some value v, which means that o is trivially on the notification path of f in $\texttt{net}(cn)$. If $v_\perp = \perp$, there is, by Future Existence, some task or some call message associated with f for some object o' in $\texttt{net}(cn)$, and, additionally, o' will have o in its forwarding list for f; hence o is on the notification path of f in this case also.

Suppose finally that $\texttt{net}(cn)$ assigns f to f', and o is on the notification path of f'. There must then be a corresponding assignment in $\mathsf{bind}\ \overline{z}.cn$, which by Future Existence yields that there is future container for the future corresponding to f with lifted value $v_\perp$. If $v_\perp \neq \perp$, then $a(f) = (v, \varepsilon)$ for some value v, which means that o is trivially on the notification path of f. If $v_\perp = \perp$, there is again, by Future Existence, some task or some call message associated with f for some object o' in $\texttt{net}(cn)$, and, additionally, o' will have o in its forwarding list for f; hence o is on the notification path of f in this case also. □

Proposition 3.9.8. *Let cn_1 and cn_2 be WF2 configurations such that $cn_1 \geqslant cn_2$. Then, if o is on the notification path of f by o_2 in n steps in cn_2, o is on the notification path of f by some o_1 in at most n steps in cn_1, such that the notification path from o to o_2 in cn_2 traverses o_1.*

Proof. Suppose $\mathsf{o}(o, a_1, u, q_{in}, q_{out}) \preceq cn_1$ and $\mathsf{o}(o, a_2, u, q_{in}, q_{out}) \preceq cn_2$. The proof is by induction on the notification path relation.

Case 1: $n = 0$, $o_2 = o$, and $\pi_1(a_2(f)) = v$. Then, we have $\pi_1(a_1(f)) = v$, and we set $n = 0$ and $o_1 = o$.

Case 2: $n = 1$, $o_2 = o$, and there is a message $\mathsf{future}(o, f, v) \preceq cn_2$. Since messages are unaffected by extension, the same message is in cn_1, and we set $n = 1$ and $o_1 = o$.

Case 3: $n = 1$, $o_2 = o$, and there is a task $\mathsf{t}(o, l, s) \preceq cn_2$, such that $l(\mathbf{ret}) = f$. Since task containers are unaffected by extension, the same container is in cn_1, and we set $n = 1$ and $o_1 = o$.

Case 4: $n = 2$, $o_2 = o$, and there is a call message $\mathsf{call}(o, o'', f, m, \overline{v}) \preceq cn_2$. Then, the same message is in cn_1, and we set $n = 2$ and $o_1 = o$.

Case 5: $n = 4$ and there is a message $\mathsf{call}(o_2, o, f, m, \overline{v}) \preceq cn_2$. Then, the same message is in cn_1, and we set $n = 3$ and $o_1 = o_2$.

Case 6: $n = n'_2 + 2$, and there is an object $\mathsf{o}(o', a'_2, u', q'_{in}, q'_{out}) \preceq cn_2$ such that $o \in \pi_2(a'_2(f))$, and o' is on the notification path of f by o_2 in n'_2 steps in cn_2. As induction hypothesis, we have that o' is on the notification path of f by some o'_1 in some number n'_1 steps ($n'_1 \leq n'_2$) in cn_1. Consider the object $\mathsf{o}(o', a'_1, u', q'_{in}, q'_{out}) \preceq cn_1$; by the definition of extended object container, we have that o is forwarding resolved for f in cn_1 at o'. This means that either $o \in \pi_2(a'_1(f))$ or $\pi_1(a_1(f)) = v$, where cn_1 assigns v to f. In the first case, we set $n = n'_1 + 2$ and $o_1 = o'_1$. In the second case, we set $n = 0$ and $o_1 = o$.

Case 7: $n = 2n_2' + n_2''$, and o is on the notification path of f' by o_2' in n_2' steps in cn_2, cn_2 assigns f to f', and o_2' is on the notification path of f by o_2 in n_2'' steps in cn_2. Since no configuration future assignments are changed by extension, we have that cn_1 assigns f to f'. As induction hypotheses, we have first that o is on the notification path of f' by some o_1' in some n_1' steps ($n_1' \leq n_2'$) in cn_1, such that the notification path from o to o_2' in cn_2 traverses o_1'. Second, we have that o_2' is on the notification path of f by some o_1'' in some n_1'' steps ($n_1'' \leq n_2''$) in cn_1. Since the path from o to o_2' in cn_2 traverses o_1', and because of forward resolution in cn_1, o_1' is on the notification path of f by o_1'' in at most $n = 2n_2' + n_1''$ steps, setting $o_1 = o_1''$. □

Proposition 3.9.9. *If cn_1 and cn_2 are WF2 configurations such that $cn_1 \geqslant cn_2$ and there are cn_1' and cn_2' such that $\mathcal{A}_2(cn_1) \rightsquigarrow cn_1'$, and $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2'$, then $cn_1' \geqslant cn_2'$.*

Proof. Let $obj = \mathsf{o}(o, a_1', u, \varepsilon, \varepsilon)$ be an object container in cn_1' and let $obj' = \mathsf{o}(o, a_2', u, \varepsilon, \varepsilon)$ be the corresponding container in cn_2'. There must then be a container $\mathsf{o}(o, a_1, u', q_{in}, q_{in}) \preceq cn_1$ and a container $\mathsf{o}(o, a_2, u', q_{in}, q_{in}) \preceq cn_2$.

Case 1: We have $a_2(x) = a_1(x)$ for all x, and $a_2(\mathbf{self}) = a_1(\mathbf{self})$. Since normalization does not affect object-environment variable-value mappings, we get that $a_2'(x) = a_1'(x)$, for all x, and $a_2'(\mathbf{self}) = a_1'(\mathbf{self})$.

Case 2: Suppose $\pi_1(a_2'(f)) = v$. Then, cn_2 must assign v to f, and o must be on the notification path of f in cn_2. Consequently, by Proposition 3.9.8, o is then on the notification path of f in cn_1, while also assigning v to f. Hence, it is the case that $\pi_1(a_1'(f)) = v$.

Case 3: Suppose $o'' \in \pi_2(a_2'(f))$. Then, f is unresolved at o in cn_2', i.e. $\pi_1(a_2'(f)) = \bot$, and thus f is unassigned in cn_2'. Consequently, f is unassigned at o in cn_2, and hence also in cn_1. We now have two cases. Assume first $o'' \in \pi_2(a_2(f))$; then, $o'' \in \pi_2(a_1(f))$ and consequently $o'' \in \pi_2(a_1'(f))$, as needed. Assume next there is some sequence of futures $f_1, f_2, \ldots, f_n$ such that cn_2 assigns f_{i+1} to f_i for $1 \leq i < n$, cn_2 assigns f to f_n, and $o'' \in \pi_2(a_2(f_1))$. By well-formedness, o is on the notification path of $f_1, f_2, \ldots, f_n$ in cn_2, and hence also in cn_1. By assumption, we have that o'' is forwarding resolved for f_1 in cn_1 at o, which means that either $o'' \in \pi_2(a_1(f_1))$ or that o'' is forwarding resolved for f_2 in cn_1 at o, etc. Hence, it will be the case that $o'' \in \pi_2(a_1'(f))$, as needed.

Case 4: Suppose $\pi_1(a_1'(f)) = v$. Then, cn_1' assigns v to f, and consequently, so does cn_2', and hence, since normalization does not affect future assignments, so does cn_1 and cn_2. If $v \in PVal$, we can then conclude. Assume instead $v = f' \in FID$. Assume $\pi_1(a_1(f)) = f'$; this means that o is on the notification path of f' in cn_1 and thus in cn_1'. Otherwise, o must be on the notification path of f in cn_1 (in some number of steps greater than 0), which means that o is also on the notification path of f in cn_1', as needed.

Case 5: Suppose $o'' \in \pi_2(a'_1(f))$. Then, f is unresolved at o in cn'_1, i.e. $\pi_1(a'_1(f)) = \bot$, and thus f is unassigned in cn'_1. Consequently, f is unassigned at o in cn_1, and hence also in cn_2. Suppose $o'' \in \pi_2(a_1(f))$. Then either $o'' \in \pi_2(a_2(f))$ or both $o'' \in \mathit{OID}(cn_2)$ and o is on the notification of f in cn_1. Assume the former; we then have $o'' \in \pi_2(a'_2(f))$ as required. Assume the latter; then we have $o'' \in \mathit{OID}(cn'_2)$ and o is still on the notification path of f in cn'_1, as needed. □

Lemma 3.9.11. *$\cong_2$ is reduction closed.*

Proof. Suppose that $cn_1 \cong_2 cn_2$ and that $\mathcal{A}_2(cn_1) \rightsquigarrow cn''_1 \geqslant cn''_2 \leftsquigarrow \mathcal{A}_2(cn_2)$. Then cn_1 and cn_2 are WF2. We proceed by case analysis on the transition $cn_1 \to cn'_1$, eliding uses of CTXT-1. For the cases T-SEND, T-RCV, MSG-SEND, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, MSG-DELAY-2, MSG-DELAY-3, CALL-RCV-2, FUT-SEND, FUT-RCV, OBJ-REG, OBJ-SEND, and OBJ-RCV, the configuration cn'_1 is fundamentally unchanged, so we set $cn'_2 = cn_2$ and have $\mathcal{A}_2(cn'_1) \rightsquigarrow cn''_1 \geqslant cn''_2 \leftsquigarrow \mathcal{A}_2(cn'_2)$.

The remaining cases include the rules for assignment and sequential control in Figure 3.3, CALL-SEND-2, RET-2, GET-2, and NEW-2. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition from cn_1 to cn'_1 of the form

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$$

where $x \in \mathtt{dom}(a)$ and $v = [\![e]\!]_{(a,l)}$. By the definition of $\geqslant$, there is a task container $\mathsf{t}(o, l, x = e; s)$ and an object container $\mathsf{o}(o, a', u', \varepsilon, \varepsilon)$, where a' has the same variable-value mappings as a, in cn''_2. Hence, it is possible to perform a transition

$$cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e; s) \to cn'\ \mathsf{o}(o, a'[v/x], u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, s)$$

and we have that

$$cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s) \cong_2 cn'\ \mathsf{o}(o, a'[v/x], u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, s)\ ,$$

setting cn'_2 to the right-hand side. This is the case since the transition does not change future-value mappings in any object container.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned} &cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s)) \\ &\quad\to\quad cn\ \mathsf{o}(o, a'_1, u_1, q_{in}, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), q_{out}))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

where $o' = [\![e_1]\!]_{(a_1,l)}$, $\overline{v} = [\![\overline{e_2}]\!]_{(a_1,l)}$, $f = \mathtt{newf}(u_1)$, and $a'_1 = \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a_1))$. By the definition of extension, we have that $\mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \preceq cn''_2$ and that there is some container $\mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn''_2$, with the same variable-value mappings as o in cn_1. Hence, we can perform a transition

$$\begin{aligned} &cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ &\quad\to\quad cn'\ \mathsf{o}(o, a'_2, u, \varepsilon, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), \varepsilon))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

where $a_2' = \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a_2))$. It now suffices to show that

$$\begin{aligned} &cn\ \mathsf{o}(o, a_1', u_1, q_{in}, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), q_{out}))\ \mathsf{t}(o, l[f/x], s) \\ \cong_2\quad &cn'\ \mathsf{o}(o, a_2', u, \varepsilon, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), \varepsilon))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

referring to the left-hand side as cn_1' and the right-hand side as cn_2'. Compared to the corresponding configurations before the transition, the only changes are that a task has progressed, a call message has been generated, and forwardings added to o' for the futures in $\overline{v}$. Since the configurations are WF2, o is on the notification path of the futures in $\overline{v}$. Hence, the changes effected by the transitions are the same for both configurations. Consequently, the equivalence holds.

RET-2: Consider a transition of the form

$$cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l, \textbf{return}\ e; s) \to cn\ \mathsf{o}(o, a_1[v/f], u_1, q_{in}, q_{out})$$

where $v = [\![e]\!]_{(a_1,l)}$ and $f = l(\textbf{ret})$. By the definition of extension, we have that $\mathsf{t}(o, l, \textbf{return}\ e; s) \preceq cn_2''$, and that there is some container $\mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn_2''$ such that $v = [\![e]\!]_{(a_2,l)}$. Hence, we can perform a transition

$$cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l, \textbf{return}\ e; s) \to cn'\ \mathsf{o}(o, a_2[v/f], u, \varepsilon, \varepsilon)$$

and we wish to prove that

$$cn\ \mathsf{o}(o, a_1[v/f], u_1, q_{in}, q_{out}) \cong_2 cn'\ \mathsf{o}(o, a_2[v/f], u, \varepsilon, \varepsilon)$$

referring to the left-hand side as cn_1' and the right-hand side as cn_2'. Note that f must be unresolved in cn_1 and cn_2, and hence cn_2'', by well-formedness. After running Algorithm 2, v will be resolved at all objects that are on the notification path of f in either configuration. Additionally, if $v \in \mathit{FID}$, the assigned value of that future, if available, will also have been distributed, and so on. The external messages resulting from both cn_1' and cn_2' will be the same, since the definition of extension does not permit the addition of forwardings to external OIDs.

GET-2: Consider a transition of the form

$$cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e.\textbf{get}; s) \to cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l[v/x], s)$$

where $[\![e]\!]_{(a_1,l)} = f$ and $\pi_1(a_1(f)) = v$. By the definition of extension and well-formedness, we have that $\mathsf{t}(o, l, x = e.\textbf{get}; s) \preceq cn_2''$ and that there exists some $\mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn_2''$ such that $[\![e]\!]_{(a_2,l)} = f$ and $\pi(a_2(f)) = v$. Hence, we can perform a transition

$$cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e.\textbf{get}; s) \to cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l[v/x], s)$$

and we have that

$$cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l[v/x], s) \cong_2 cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l[v/x], s)$$

setting cn_2' to the right-hand side. In the same way as for WFIELD-2, this is the case since the transition does not change the future-value mappings in any object container.

NEW-2: Consider a transition of the form

$$\begin{aligned} & cn\ \mathtt{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathtt{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \\ & \quad \rightarrow \quad cn\ \mathtt{o}(o, a_1', u_1, q_{in}, q_{out})\ \mathtt{t}(o, l[o'/x], s)\ \mathtt{o}(o', a', u_1, \varepsilon, \varepsilon) \end{aligned}$$

where it is the case that $o' = \mathtt{newo}(u_1)$, $\overline{v} = [\![\overline{e}]\!]_{(a_1,l)}$, $a' = \mathtt{init}(\overline{v}, \mathtt{init}(C, \overline{v}, o'))$, and $a_1' = \mathtt{fw}(\overline{v}, o', a_1)$. By the definition of extension, we have that $\mathtt{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \preceq cn_2''$ and that there is some $\mathtt{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn_2''$ such that $[\![\overline{e}]\!]_{(a_2,l)} = \overline{v}$. We can then perform a transition

$$\begin{aligned} & cn'\ \mathtt{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathtt{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \\ & \quad \rightarrow \quad cn'\ \mathtt{o}(o, a_2', u, \varepsilon, \varepsilon)\ \mathtt{t}(o, l[o'/x], s)\ \mathtt{o}(o', a', u, \varepsilon, \varepsilon) \end{aligned}$$

where $a_2' = \mathtt{fw}(\overline{v}, o', a_2)$. We wish to prove that

$$\begin{aligned} & cn\ \mathtt{o}(o, a_1', u_1, q_{in}, q_{out})\ \mathtt{t}(o, l[o'/x], s)\ \mathtt{o}(o', a', u_1, \varepsilon, \varepsilon) \\ & \quad \cong_2 \quad cn'\ \mathtt{o}(o, a_2', u, \varepsilon, \varepsilon)\ \mathtt{t}(o, l[o'/x], s)\ \mathtt{o}(o', a', u, \varepsilon, \varepsilon) \end{aligned}$$

referring to the left-hand side as cn_1' and the right-hand side as cn_2'. Note that the only added forwardings through the transition are from o to o', and that they will be for the same futures in both cn_1' and cn_2', by well-formedness. Hence, after running Algorithm 2, there will be an object container $\mathtt{o}(o', a'', u, \varepsilon, \varepsilon)$ in both resulting configurations, where all futures that are assigned in the original configurations are resolved. In addition both resulting configurations will contain the task $\mathtt{t}(o, l[o'/x], s)$. It remains to argue that the object container with OID o in the configuration resulting from cn_1' extends the corresponding object container resulting from cn_2'. The only difference between the containers compared to their counterparts in cn_1'' and cn_2'' is the possible addition of o' in some forwarding lists; however, these additions will be the same for both containers, so extension still holds.

The same arguments are valid with minor changes for the case when $cn_2'' \geqslant cn_1''$. $\square$

Lemma 3.9.12. $\cong_2$ *is context closed.*

Proof. Suppose $cn_1 \cong_2 cn_2$, and let cn be a context configuration. We then have that cn_1 and cn_2 are WF2.

Assume $cn_1 \geqslant cn_2$ and that $cn_1\ cn$ is WF2. We first show that $cn_2\ cn$ is WF2. For OID Uniqueness, note that $OID(cn_1) = OID(cn_2)$ by the definition of extension; hence, any OID clash violates the well-formedness of $cn_1\ cn$. For Task-Object Existence, it suffices to consider the case $\mathtt{t}(o, l, s) \preceq cn$ and $\mathtt{o}(o, a, u, q_{in}, q_{out}) \preceq cn_1$); by the definition of $\cong_2$, there is then also an object container with OID o. Object-Node Existence follows from $\mathtt{graph}(cn_1) = \mathtt{graph}(cn_2)$, which holds by the definition of $\cong_2$. Buffer Cleanliness distributes over composition and thus holds for cn_1, cn and cn_2 separately. Local Routing Consistency only concerns nodes, which are unaffected by adding the context cn. Call Uniqueness holds since messages in $cn_2\ cn$ are either from cn, correspond to some message or task (which is a single writer) in cn_1; Single Writer holds for similar reasons. External OID follows again by noting that nodes are unchanged when applying cn.

For Future Liveness, assume that $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_2\ cn$, and either f is active for o in $cn_2\ cn$, $a(f) \downarrow$, or $cn_2\ cn$ assigns f to f' and o is on the notification path of f'. Assume without loss of generalization that $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, since the property already holds in cn_2 separately. We consider the cases in turn. Suppose f is active for o in $cn_2\ cn$; then f is also active for o in $cn_1\ cn$, so o is on the notification path of f in $cn_1\ cn$. If the path in $cn_1\ cn$ from which o can receive f traverses some $o' \in OID(cn_1)$, there must be an equivalent path in $cn_2\ cn$, since extension does not change forwardings to configuration-external OIDs. The case where the path does not go outside of cn is even simples. The other cases hold for similar reasons.

The converse of this well-formedness argument, that $cn_1\ cn$ is WF2 whenever $cn_2\ cn$ is WF2, holds with only minor changes to the above reasoning.

Assume that $\mathcal{A}_2(cn_1) \rightsquigarrow cn'_1 \geqslant cn'_2 \leftsquigarrow \mathcal{A}_2(cn_2)$. Let $\mathcal{A}_2(cn_1\ cn) \rightsquigarrow cn''_1$ and $\mathcal{A}_2(cn_2\ cn) \rightsquigarrow cn''_2$. Well-formedness holds from the above reasoning. We wish to prove that $cn''_1 \geqslant cn''_2$. We consider the conditions in turn.

$\mathtt{graph}(cn''_1) = \mathtt{graph}(cn_1\ cn) = \mathtt{graph}(cn_1) = \mathtt{graph}(cn_2) = \mathtt{graph}(cn_2\ cn) = \mathtt{graph}(cn''_2)$.

$\mathsf{m}(cn''_1) = \mathsf{m}_2(cn_1\ cn) = \mathsf{m}_2(cn_1) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2\ cn) = \mathsf{m}(cn''_2)$.

Since $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$, we have $\mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn)$. Hence, $\mathsf{t}(cn''_1) = \mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn) = \mathsf{t}(cn''_2)$.

Consider $obj = \mathsf{o}(o, a_1, u, \varepsilon, \varepsilon) \preceq cn''_1$ and $obj' = \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn''_2$. We want to argue that obj extends obj'. Suppose the containers are derived from an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$.

Case 1: Since Algorithm 2 does not affect **self** or stored values, we have $a_2(\mathbf{self}) = a_1(\mathbf{self})$ and $a_2(x) = a_1(x)$ for all x.

Case 2: Assume $\pi_1(a_2(f)) = v$. Then, if $\pi_1(a(f)) = v$, we also have $\pi_1(a_1(f)) = v$. Suppose $\pi_1(a(f)) = \bot$; then, o is on the notification path of f in $cn_2\ cn$, and $cn_2\ cn$ assigns v to f. If the notification path traverses $o' \in OID(cn)$, a corresponding notification path will traverse o' in $cn_1\ cn$, since the forwardings and external future messages of cn_1 and cn_2 to external objects coincide after running Algorithm 2. Consequently, o is also on the notification path of f in $cn_1\ cn$, and we have $\pi(a_1(f)) = v$ by well-formedness and the definition of Algorithm 2.

Case 3: Assume $o'' \in \pi_2(a_2(f))$. Then, f is unresolved in $cn_1\ cn$ and $cn_2\ cn$. We need to argue that o'' is forwarding resolved for o in cn''_1. Suppose that $o'' \in \pi_2(a(f))$; then clearly $o'' \in \pi_2(a_1(f))$. Suppose $o'' \notin \pi_2(a(f))$; then there must some sequence of futures $f_1, \ldots, f_n$, such that $cn_1\ cn$ and $cn_2\ cn$ assign f_i to f_{i+1} for $1 \leq i < n$, assign f to f_n, and additionally $o'' \in \pi_2(a(f_1))$ and o is on the notification path of f_1. After running Algorithm 2, we will clearly have $o'' \in \pi_2(a_1(f))$.

Case 4: Assume $\pi_1(a_1(f)) = v$. Then, either cn_1 or cn assigns v to f, which implies that either cn_2 or cn assigns v to f. Suppose $v = f' \in \mathit{FID}$. If $\pi_1(a(f)) = f'$, o is on the notification path of f' in cn_1'' by well-formedness.

Case 5: Assume $o'' \in \pi_2(a_1(f))$. Then f is unassigned and on the notification path of o in $cn_1\ cn$ and consequently also in $cn_2\ cn$. Since external forwardings and messages are the same in cn_1 and cn_2, the forwarding for o'' was added either by some internal message in cn or by a message or forwarding present in both cn_1 and cn_2. Consequently, $o'' \in \pi_2(a_2(f))$. □

Proposition 3.9.13. $\cong_2$ *is a type 2 witness relation.*

Proof. By Lemma 3.9.11 and Lemma 3.9.12, it suffices to prove barb preservation in both directions. Assume without loss of generalization that $\mathcal{A}(cn_1) \rightsquigarrow cn_1' \geqslant cn_2' \leftsquigarrow \mathcal{A}(cn_2)$. Suppose $cn_1 \downarrow obs$, where $obs = ext!m(\overline{v})$; then $msg \preceq cn_1$ with $msg = \mathsf{call}(ext, o, f, m, \overline{v})$, whence $msg \preceq cn_1'$. By the definition of extension, we then have $msg \preceq cn_2'$, such that msg is in some link queue. It is then possible to repeatedly use the rule MSG-DELAY-3 to reach a configuration cn_2'' where msg is at the top of that link queue. We thus have $cn_2 \rightarrow^* cn_2' \rightarrow^* cn_2'' \downarrow obs$, and conclude that $cn_2 \Downarrow obs$. The proof of the converse property is symmetric. □

Lemma 3.9.14. *Suppose that cn' is WF2, and $cn \geqslant cn'$. Then, cn is WF2 as well, and $cn \simeq_2 cn'$.*

Proof. It suffices to prove that cn is WF2, since we then have that $cn \cong_2 cn'$ by Proposition 3.9.9, which by Proposition 3.9.13 implies that $cn \simeq_2 cn'$. Note that every WF2 clause except Single Writer and Future Liveness holds straightforwardly in cn from the assumption that cn' is WF2, since only future maps in object containers are different. For Single Writer, note that cn does not introduce mappings for unused futures, and does not introduce new messages. For Future Liveness, assume some f is active for o in cn; then f is active for o in cn', and therefore, o is on the notification path of f in cn'. Then, since cn does not remove any OIDs from forwarding lists or any instantiations of futures, o must still be on the notification path of in cn (possibly with fewer steps). □

Lemma 3.9.15. *Let* $\mathsf{bind}\ \overline{z}.cn$ *be a WF1 configuration in standard form.*

1. *If* $\mathsf{bind}\ \overline{z}.cn \rightarrow \mathsf{bind}\ \overline{z}'.cn'$, *then for some* cn'', $\mathtt{net}(cn) \rightarrow^* cn'' \cong_2 \mathtt{net}(cn')$.

2. *If* $\mathtt{net}(cn) \rightarrow cn''$, *then for some* $\overline{z}'$ *and* cn', $\mathsf{bind}\ \overline{z}.cn \rightarrow^* \mathsf{bind}\ \overline{z}'.cn'$ *and* $cn'' \cong_2 \mathtt{net}(cn')$.

Proof. 1. We proceed by cases on the nature of the given type 1 transition. Let

$$\mathsf{bind}\ \overline{z}.cn \rightarrow \mathsf{bind}\ \overline{z}'.cn'.$$

Fix cn_{graph} and a name representation map `rep`. As above we elide uses of CTXT-1 in both semantics by applying the rules to arbitrary configuration subsets, and we elide uses of CTXT-2 in the type 1 semantics, by considering transitions in arbitrary binding contexts. Each of the remaining transitions in Figure 3.1 immediately translates into a corresponding transition at type 2 level, and moreover, the resulting type 2 configuration is in normal form.

Consider for instance rule WFIELD. We obtain a type 1 transition of the form

$$\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=e;s) \rightarrow \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a[v/x])\ \mathsf{t}(o,l,s)$$

where $x \in \mathtt{dom}(a)$ and $v = [\![e]\!]_{(a,l)}$. Let $v' = [\![e]\!]_{(\mathtt{rep}(a),\mathtt{rep}(l))}$; then $\mathtt{rep}(v) = v'$ by Equation 3.1. We obtain:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=e;s))} \\
= & (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ \mathtt{net}(\mathsf{t}(o,l,x=e;s),\mathtt{rep}))(cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})(\mathtt{net}(\mathsf{t}(o,l,x=e;s),\mathtt{rep})(cn_{graph}))) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})(\mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=e;s)\ cn_{graph})) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=e;s)\ cn_{graph}) \\
\rightarrow & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a)[v'/x],u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a[v/x]),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a[v/x])\ \mathsf{t}(o,l,s))
\end{array}$$

using the rule WFIELD-2 to derive the transition.

Similar rules are proved in the same way. We turn to the remaining rules.

CALL-SEND: Consider the following type 1 transition:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=e_1!m(\overline{e_2});s)} \\
\quad\rightarrow & \mathsf{bind}\ f\,\overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[f/x],s)\ \mathsf{f}(f,\perp)\ \mathsf{c}(o',f,m,\overline{v})
\end{array}$$

where $o' = [\![e_1]\!]_{(a,l)}$ and $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$. We calculate:

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = e_1!m(\overline{e_2});s))} \\
= & (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ \\
 & \mathtt{net}(\mathsf{t}(o,l,x = e_1!m(\overline{e_2});s),\mathtt{rep}))(cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \\
 & (\mathtt{net}(\mathsf{t}(o,l,x = e_1!m(\overline{e_2});s),\mathtt{rep})(cn_{graph}))) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \\
 & (\mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x = e_1!m(\overline{e_2});s)\ cn_{graph})) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon) \\
 & \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x = e_1!m(\overline{e_2});s)\ cn_{graph}) \\
\to \circ \cong_2 & \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{fw}(\mathtt{rep}'(\overline{v}),\mathtt{rep}'(o'),\mathtt{init}(f',\mathtt{rep}'(a))), \\
 & u_0,\varepsilon,\mathsf{enq}(\mathsf{call}(\mathtt{rep}'(o'),\mathtt{rep}'(o),f',m,\mathtt{rep}'(\overline{v})),\varepsilon)) \\
 & \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[f'/x],s)\ cn_{graph}) \\
\equiv_2 & \mathtt{net}(cn,\mathtt{rep}')(\mathsf{send}(\mathsf{call}(\mathtt{rep}'(o'),\mathtt{rep}'(o'),f',m,\mathtt{rep}'(\overline{v})), \\
 & \mathsf{o}(\mathtt{rep}'(o),\mathtt{fw}(\mathtt{rep}'(\overline{v}),\mathtt{rep}'(o'),\mathtt{init}(f',\mathtt{rep}'(a))),u_0,\varepsilon,\varepsilon) \\
 & \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[f'/x],s)\ cn_{graph})) \\
= & \mathtt{net}(cn,\mathtt{rep}')(\mathsf{send}(\mathsf{call}(\mathtt{rep}'(o'),\mathtt{rep}'(o'),f',m,\mathtt{rep}'(\overline{v})), \\
 & \mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[f'/x],s)\ cn_{graph})) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[f/x],s)\ \mathsf{f}(f,\bot)\ \mathsf{c}(o',f,m,\overline{v}))
\end{array}$$

using CALL-SEND-2, and where $f' = \mathtt{newf}(u_0)$ and $\mathtt{rep}' = \mathtt{rep}[f'/f]$.

CALL-RCV: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{c}(o,f,m,\overline{v}) \to \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s)$$

where $l = \mathtt{locals}(o,f,m,\overline{v})$ and $s = \mathtt{body}(o,m)$. We calculate:

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{c}(o,f,m,\overline{v}))} \\
= & (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ \mathtt{net}(\mathsf{c}(o,f,m,\overline{v}),\mathtt{rep}))(cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon) \\
 & \mathsf{send}(\mathsf{call}(\mathtt{rep}(o),\mathtt{rep}(o),\mathtt{rep}(f),m,\mathtt{rep}(\overline{v})),cn_{graph})) \\
\equiv_2 & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s))
\end{array}$$

RET: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,\mathbf{return}\ e;s)\ \mathsf{f}(f,\bot) \to \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)$$

where $l(\mathbf{ret}) = f$ and $v = [\![e]\!]_{(a,l)}$. We calculate:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,\mathbf{return}\ e;s)\ \mathsf{f}(f,\perp))} \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon) \\
 & \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),\mathbf{return}\ e;s)\ cn_{graph}) \\
\to\circ\cong_2 & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a)[\mathtt{rep}(v)/\mathtt{rep}(f)],u_0,\varepsilon,\varepsilon)\ cn_{graph}) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v))
\end{array}$$

using RET-2.

GET: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l,x=e.\mathbf{get};s) \to \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l[v/x],s)$$

where $f = [\![e]\!]_{(a,l)}$. We calculate:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l,x=e.\mathbf{get};s))} \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=e.\mathbf{get};s)\ cn_{graph}) \\
\to & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l)[\mathtt{rep}(v)/x],s)\ cn_{graph}) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l[v/x],s))
\end{array}$$

using GET-2.

NEW: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=\mathbf{new}\ C(\overline{e});s) \to \mathsf{bind}\ o'\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a')$$

where $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$ and $a' = \mathtt{init}(C,\overline{v},o')$. We calculate:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=\mathbf{new}\ C(\overline{e});s))} \\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon) \\
 & \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=\mathbf{new}\ C(\overline{e});s)\ cn_{graph}) \\
\to\circ\cong_2 & \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[o''/x],s) \\
 & \mathsf{o}(o'',\mathtt{rep}'(a'),u_0,\varepsilon,\varepsilon)\ cn_{graph}) \\
= & \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l[o''/x]),s) \\
 & \mathsf{o}(o'',\mathtt{rep}'(a'),u_0,\varepsilon,\varepsilon)\ cn_{graph}) \\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a'))
\end{array}$$

where $\mathtt{rep}' = \mathtt{rep}[o''/o']$ and we use Equation 3.1 as usual. This completes the proof of property 1.

2. We proceed now by cases on the type 2 transition. Suppose $\mathtt{net}(cn) \to cn''$, and we find $\overline{z}'$, cn' to complete the diagram as stated in the lemma. As above, we apply the rules to configuration subsets, to elide uses of CTXT-1. Rules in Figure 3.3

are straightforward and omitted. For the rules in Figure 3.4 excepting CALL-SEND-2, RET-2, GET-2, and NEW-2, we can choose $\overline{z}' = \overline{z}$ and $cn' = cn$, since then by Corollary 3.8.18, $\mathtt{net}(cn) \equiv_2 cn''$.

For the four remaining cases, each case is obtained by reversing the arguments, i.e., proving that if the type 2 transition holds, depending on the rule application and shape of configurations, then also the corresponding type 1 transition holds, and the resulting pair of configurations are in $\cong_2$. This completes the argument. $\square$

Theorem 3.9.16. *For all well-formed type 1 configurations* $\mathsf{bind}\ \overline{z}.cn$ *in standard form,* $\mathsf{bind}\ \overline{z}.cn \ \simeq \mathtt{net}(cn)$.

Proof. We exhibit a conflated witness relation $\mathcal{R}$ defined as

$$\mathcal{R} = \{(\mathsf{bind}\ \overline{z}.cn, cn') \mid \mathtt{net}(cn) \cong_2 cn'\}\ ,$$

where $\mathsf{bind}\ \overline{z}.cn$ is a WF1 configuration in standard form, and cn' is a WF2 configuration. Since the identity relation is included in $\cong_2$, we have $(\mathsf{bind}\ \overline{z}.cn, \mathtt{net}(cn)) \in \mathcal{R}$. We show that $\mathcal{R}$ is a conflated type 1 and type 2 witness relation. Via Lemma 3.9.11, $\cong_2$ is reduction closed.

Suppose $\mathsf{bind}\ \overline{z}.cn_1\ \mathcal{R}\ cn_2$ (or the converse for $\mathcal{R}^{-1}$); then $\mathsf{bind}\ \overline{z}.cn_1$ is WF1 and in standard form, cn_2 is WF2, and $\mathtt{net}(cn_1) \cong_2 cn_2$.

For reduction closure, assume $\mathsf{bind}\ \overline{z}.cn_1 \to \mathsf{bind}\ \overline{z}'.cn_1'$, where $\mathsf{bind}\ \overline{z}'.cn_1'$ is in standard form. Then, by property 1 of Lemma 3.9.15, $\mathtt{net}(cn_1) \to^* cn_1'' \cong_2 \mathtt{net}(cn_1')$. This means that, for some cn_2', $cn_2 \to^* cn_2'$ and $cn_2' \cong_2 cn_1''$. Hence, by transitivity of $\cong_2$, $\mathsf{bind}\ \overline{z}'.cn_1'\ \mathcal{R}\ cn_2'$. For converse reduction closure, assume $cn_2 \to cn_2'$. Then, $\mathtt{net}(cn_1) \to^* cn_2''$ and $cn_2' \cong_2 cn_2''$. By property 2 of Lemma 3.9.15, this means that $\mathsf{bind}\ \overline{z}.cn \to^* \mathsf{bind}\ \overline{z}'.cn_1'$ and $cn_2'' \cong_2 \mathtt{net}(cn_1')$. Hence, by the transitivity of $\cong_2$, $cn_2'\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{z}'.cn_1'$.

For context closure, assume $\mathsf{bind}\ \overline{z}'.cn_1\ \ cn$ is WF1 and in standard form, and consider the configuration $\mathtt{net}(cn, \mathtt{rep})(cn_2)$, which in effect applies cn to cn_2. We first need to show that this configuration is WF2. Object-Node Existence holds, since by the definition of $\mathtt{net}$, all objects in cn become attached to a node in cn_2. Buffer Cleanliness and Local Routing Consistency also hold by the definition of $\mathtt{net}$. For OID Uniqueness, it suffices to consider the case when $\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn_2$ and $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathtt{net}(cn, \mathtt{rep})(cn_2)$ where $\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \not\preceq \mathtt{net}(cn, \mathtt{rep})(cn_2)$; then, if $o_1 = o_2$, there is a corresponding clash in $\mathsf{bind}\ \overline{z}'.cn_1\ \ cn$, violating WF1. For Task-Object Existence, assume $\mathsf{t}(o, l, s) \preceq \mathtt{net}(cn, \mathtt{rep})(cn_2)$ and $\mathsf{t}(o, l, s) \not\preceq cn_2$; then if there is no object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq \mathtt{net}(cn, \mathtt{rep})(cn_2)$, there is no corresponding object container in $\mathsf{bind}\ \overline{z}'.cn_1\ \ cn$, violating WF1. For Call Uniqueness, it suffices to consider the case when $\mathsf{call}(o_1, o_1', f_1, m_1, \overline{v}_1) \preceq cn_2$ and $\mathsf{call}(o_2, o_2', f_2, m_2, \overline{v}_2) \preceq \mathtt{net}(cn, \mathtt{rep})(cn_2)$ where $\mathsf{call}(o_1, o_1', f_1, m_1, \overline{v}_1) \not\preceq cn_2$; then, if $f_1 = f_2$, there is a corresponding clash between messages in $\mathsf{bind}\ \overline{z}'.cn_1\ \ cn$, which goes against the WF1 assumption. If Future Uniqueness does not hold, this means that there are duplicate future containers in $\mathsf{bind}\ \overline{z}'.cn_1\ cn$, or that cn_2 assigns some future different values, both of which are ruled out. For Single Writer, a clash of identifiers in return futures for tasks and calls can again be traced back to a clash in $\mathsf{bind}\ cn_1\ cn$, contradicting WF1. External OID holds since *ext* cannot be defined in WF1 configurations, and routing tables are intact from cn_2. For Future Liveness, if f is active for o in $\mathtt{net}(cn, \mathtt{rep})(cn_2)$, a corresponding future is active in $\mathsf{bind}\ \overline{z}'.cn_1\ cn$, which either has a future container with a value, or a task that is producing it. Since cn_2 is WF2, and $\mathtt{net}$ adds forwarding for futures, o will be on the notification of f. The same argument holds when o is on the notification path of the future f' which is assigned f.

It remains to show that $\mathsf{bind}\ \overline{z}.cn_1\ cn\ \mathcal{R}\ \mathtt{net}(cn, \mathtt{rep})(cn_2)$. The network graphs of $\mathtt{net}(cn_1\ cn)$ and $\mathtt{net}(cn, \mathtt{rep})(cn_2)$ coincide, since applying cn does not introduce any nodes or links. Clearly, if and only if a task or a message is in cn, a corresponding task or message is introduced in $\mathtt{net}(cn, \mathtt{rep})(cn_2)$. We already have that the remaining tasks and tasks spawned from messages in cn_1 correspond to those in cn_2. As for external messages, they are either newly introduced via the context, and are thus in both composed configurations, or they come from the original configuration. With respect to objects, the future maps of objects in the normal form of $\mathtt{net}(cn_1\ cn)$ are possibly extended when compared to future maps for objects in the normal form of $\mathtt{net}(cn, \mathtt{rep})(cn_2)$, but in all other ways, objects in the normal forms are equal. Hence, $\mathtt{net}(cn_1\ cn) \cong_2 \mathtt{net}(cn, \mathtt{rep})(cn_2)$.

For converse context closure, assume $cn_2\ cn$ is WF2, and apply cn to produce $\mathsf{bind}\ \overline{z}.\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$ in standard form. We first need to show that this configuration is WF1. Let cn' be the multiset difference of $\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$ and cn_1. For OID Uniqueness, it suffices to consider the case where $\mathsf{o}(o_1, a_1) \preceq cn_1$ and $\mathsf{o}(o_2, a_2) \preceq cn'$; then, if $o_1 = o_2$, there is a corresponding clash in $cn_2\ cn$, violating WF2. For Task-Object Existence, assume $\mathsf{t}(o, l, s) \preceq cn'$; then, if there is no object container $\mathsf{o}(o, a) \preceq cn_1 cn'$, this violates WF2 for the task corresponding to $\mathsf{t}(o, l, s)$ in $cn_2\ cn$. For Call Uniqueness, it suffices to consider the case where $\mathsf{c}(o_1, f_1, m_1, \overline{v}_1) \preceq cn_1$ and $\mathsf{c}(o_2, f_2, m_2, \overline{v}_2) \preceq cn'$; note that if $f_1 = f_2$, there would have been a WF2 Call Uniqueness violation in $cn_2\ cn$. For Single Writer, clashes in future identifiers can be traced back similarly to clashes in $cn_2\ cn$, violating WF2. For Future Existence, if f is active for o in $cn_1\ cn'$, a corresponding future is active in $cn_2\ cn$, meaning that by the WF2 property, there is a notification path to an assignment, task or message, which translates to there being a future container, and a call container or task container when appropriate in $cn_1\ cn'$.

It remains to show that $cn_2\ cn\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{z}'.\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$. Again, let cn' be the multiset difference of $\mathtt{ten}(cn, \mathtt{rep}^{-1})(cn_1)$ and cn_1. The network graphs of $\mathtt{net}(cn_1\ cn')$ and $cn_2\ cn$ coincide, since cn does not introduce any new nodes or links. Clearly, if and only if a task or non-external message is in cn, a corresponding task or message is in cn'. We already have that the remaining tasks and tasks corresponding to messages in cn_1 and cn_2 coincide. As for external messages, either they are newly introduced via the context, and are thus in both composed configurations, or they come from the original configuration, and thus coincide. With respect to objects, the future maps of objects in the normal form of $\texttt{net}(cn_1\ cn')$ are possibly extended when compared to future maps for objects in the normal form of $cn_2\ cn$, but in all other ways, objects in the normal forms are equal. Hence, $\texttt{net}(cn_1\ cn') \cong_2 cn_2\ cn$.

For barb preservation, assume $\mathsf{bind}\ \overline{z}.cn_1 \downarrow obs$. Then $\texttt{net}(cn_1) \Downarrow obs$, which by normal form equivalance yields $cn_2 \Downarrow obs$, as needed. For converse barb preservation, assume $cn_2 \downarrow obs$. Then, by normal form equivalence, $\texttt{net}(cn_1) \Downarrow obs$, and consequently $cn_1 \downarrow obs$, whereby $cn_1 \Downarrow obs$. $\square$

Chapter 4

ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects

Karl Palmskog[1] Mads Dam[1] Andreas Lundblad[1] Ali Jafari[2]

[1]KTH Royal Institute of Technology, Sweden
{palmskog, mfd, landreas}@kth.se
[2]Reykjavik University, Iceland
ali11@ru.is

Abstract

We present a formalized, fully decentralized runtime semantics for a core subset of ABS, a language and framework for modeling distributed object-oriented systems. The semantics incorporates an abstract graph representation of a network infrastructure, with network endpoints represented as graph nodes, and links as arcs with buffers, corresponding to OSI layer 2 interconnects. The key problem we wish to address is how to allocate computational tasks to nodes so that certain performance objectives are met. To this end, we use the semantics as a foundation for performing network-adaptive task execution via object migration between nodes. Adaptability is analyzed in terms of three Quality of Service objectives: node load, arc load and message latency. We have implemented the key parts of our semantics in a simulator and evaluated how well objectives are achieved for some application-relevant choices of network topology, migration procedure and ABS program. The evaluation suggests that it is feasible in a decentralized setting to continually meet both the objective of a node-balanced task allocation and make headway towards minimizing communication, and thus arc load and message latency.

4.1 Introduction

An important problem, made more relevant by recent interest in cloud computing, is how to decouple computational processes from the underlying physical infrastructure on which they execute. One motivation for such decoupling is to free applications from handling resource allocation issues, which can instead be taken care of in a provably correct and transparent fashion using generic, application-independent mechanisms. Potentially, tasks can then be performed at the physical machine most suited at the moment, continually meeting global system requirements such as utilization and power consumption, or task-local requirements such as a response time.

We consider the problem of runtime adaptation of tasks in the context of a core subset of ABS [103], a language for modeling distributed object-oriented systems developed in the EU FP7 HATS project. In ongoing work, described in Chapter 2 and Chapter 3, we are developing network-aware semantics for different fragments of ABS with some novel features. Specifically, we let objects execute on network nodes connected point-to-point using asynchronous message passing links, and show how location independent routing in such a setting can be used to support efficient, transparent, and robust (lock-free) object migration. Here, we examine how adaptation can be performed in such a model by a controller process running on each node.

To enable precise reasoning and experiments on adaptability, we define three central Quality of Services (QoS) objectives against which a solution for runtime adaptation in our context can be assessed: node load, arc load and message latency. We abstract from many practical, implementation-level concerns when interpreting these objectives in our setting. The load for a node is the number of active tasks running on it. The load for an arc is the number of messages traversing the arc. The latency for a message is the number of hops needed to reach its destination. We then restrict our consideration of adaptability to the problem of how and when to migrate objects to achieve the objectives as well as possible, given a specific network topology, ABS program, and node-local procedure for managing migrations. Using a simulator which implements the key parts of our semantics, we have investigated how well objectives are met for some application-relevant choices of network topologies, programs and migration procedures.

Section 4.2 and 4.3 describe the ABS language and our novel ABS-NET semantics for execution of ABS programs in a network. Section 4.4 describes our approach to runtime adaptation via object migration. Section 4.5 describes the simulator, our benchmark scenarios, and simulation results; Section 4.6 concludes.

4.2 ABS Background

ABS [77] is a language and framework for modeling distributed object-oriented systems, developed in the EU FP7 HATS project. Core ABS [103] is a language which contains the main features of ABS: a functional level for expressing data structures and side-effect free internal computations of objects, and an object level for expressing concurrent objects, and communication among such objects via method invocation. The object level defines syntax—reminiscent of Java's—for interfaces, classes, methods, object creation and method calls. The object level is accompanied by a type system and an operational semantics which preserves well-typing. One consequence of well-typing is that many runtime errors are ruled out for type-checked programs; when an object makes a call to a method m using an object identifier o, there always exists an object associated with o, which is an instance of a class which implements m. ABS uses placeholders in the form of futures for the result of method calls, allowing a caller to avoid blocking until the result value is actually required. In the variant of Core ABS we consider, a single object is the unit of concurrency, as in the variant of Albert et al. [7]. This means that objects at runtime can be viewed as actors, communicating between themselves only via asynchronous message passing.

An example of an ABS interface with implementing classes is given below. The `CastNode` interface defines a method `aggregate`, which, when called on an object, performs a convergecast operation in the object-reference binary tree rooted at that object. Specifically, this means that if an object implementing `CastNode` is a leaf in the tree (an instance of `LeafNode`), it simply returns a locally known integer, but if the object has child nodes in the tree (an instance of `BranchNode`), `aggregate` is called on both of those objects and the results are added to the local integer and returned. In this way, the `aggregate` method for the object o always returns the aggregate (sum) of all local values in the binary tree of objects rooted at o. The variables `fLeft` and `fRight` in the implementation of `aggregate` hold the placeholders (futures) for integers that result from the asynchronous method calls. The values are then retrieved through the **.get** operator, which can cause blocking until the method call has finished.

```
interface CastNode {        class LeafCastNode(Int val) implements CastNode {
    Int aggregate();            Int aggregate() { return val; }
}                           }
```

```
class BranchCastNode(Int val, CastNode left, CastNode right)
  implements CastNode {
   Int aggregate() {
      Fut<Int> fLeft = left!aggregate();
      Fut<Int> fRight = right!aggregate();
      Int aggregateLeft = fLeft.get;
      Int aggregateRight = fRight.get;
      return val + aggregateLeft + aggregateRight;
    }
}
```

4.3 Network Model and Semantics

To reason about object adaptability with respect to environmental conditions, we bring selected parts of the infrastructure of a distributed system into our model, namely, network endpoints and links. Endpoints and links are modelled as graph nodes and arcs with FIFO-ordered message queues, respectively. Conceptually, a node consists of an interpreter layer, where local objects reside, and a node controller, which acts as a mediator between the environment and node-local objects, as illustrated in Figure 4.1. The dashed arrow in the figure signifies that the identifier of object o_2 is known by object o_1, allowing o_1 to send method invocations to o_2. The structure is similar to that used in other programming-language oriented distributed system models, e.g., a proposed semantics for future Erlang [192]. Here, the node controller also contains logic for decision-making on adaptability. Seen abstractly, adaptability in this context becomes the problem of deciding when and where to migrate objects to achieve the QoS objectives—with the added constraint that all reallocations must be decided locally at each node.

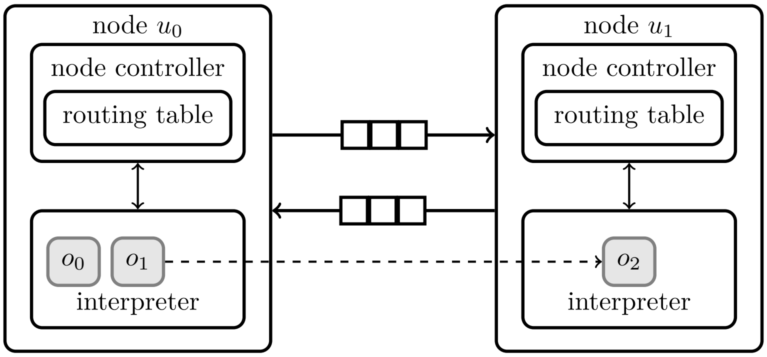


Figure 4.1: Nodes, node controllers, and interpreter layers

To achieve location transparency, the basic problem is to route messages correctly between objects that have no prior, mutual knowledge of where they are located. Many solutions have been examined in the literature, including centralized or decentralized location servers, pointer chaining, and broadcast or multicast search. Sewell et al. [184] discuss many of these solutions, and their relative merits.

We are developing a novel approach to location transparency based on location-independent (also called name-independent) routing, as first outlined in Chapter 2, where the idea is to defer the maintenance of message routes to an explicit routing process executing independently of application-level messaging. Adapted to the approach suggested here, a node controller executing on each network node is responsible for maintaining routing information by exchanging routing tables with adjacent nodes in the network. This allows object migration to be supported in a transparent fashion with only modest extension to the runtime state.

We have defined a new operational semantics of Core ABS programs, in the same rewriting logic style [40] as the standard semantics, that characterizes task execution of objects located on, and moving between, network nodes. Adaptability features such as routing table exchange and object migration are modelled as nondeterministic events, with the node controller consisting of nothing more than a globally unique identifier and a routing table. We refer to the combination of the Core ABS functional layer, Core ABS object syntax, and our novel operational semantics as ABS-NET. We intend for the semantics to both guide implementation, by defining a baseline for retaining program runtime behavior similar to Core ABS in a networked, decentralized setting, and provide opportunities for further theoretical analysis of specific adaptability strategies by refinement.

A complete description of the syntax and formal semantics of Core ABS and ABS-NET is available in Appendix A and Appendix B, respectively, with semantic equivalence explored in Chapter 2 and Chapter 3. Below, we give an overview of the ABS-NET network model and semantics, with details on node controller behavior, which is to an extent agnostic towards the underlying actor environment.

4.3.1 Runtime Configurations

The node controller's relationship with the interpreter layer residing on the node is symbiotic. On one hand, the node controller provides message delivery services and callback functions to obtain new globally unique object identifiers for objects residing in the interpreter layer. On the other hand, the node controller triggers object movement by using callback functions that the interpreter layer makes available. We assume a node controller is aware of the asynchronous links through which it can communicate via message passing with other node controllers. In essence, the aim is that node controllers should be realizable on top of a network with only OSI layer 2 interconnects, meaning that the required primitives for computation and communication can be implemented directly in hardware with high performance. If this is the case, running controllers on top of overlays using higher-layer interconnects such as sockets is also feasible, which is what we do in our simulator.

The global state in ABS-NET is formally a pair $\{net\}\,\{cn\}$. The network part $\{net\}$ is a set of nodes and arcs. In a node $\mathsf{nd}\,(u, \tau)$, u is a node identifier (assumed globally unique) and τ is a routing table, used to route object-related messages in the proper direction. In an arc $\mathsf{ar}\,(u, Q, u')$, representing a unidirectional link from u to u', Q is a FIFO-ordered queue of messages. The other part of the global state, $\{cn\}$, is a set of objects, with each object implicitly attached to a node in the network, and able to send and receive messages with the assistance of its host. A message msg can be (1) a table message $\textsc{Table}\,(\tau)$, used to pass a routing table τ from one node to another to update local routes, (2) an object message $\textsc{Object}\,(\mathit{object})$ containing a complete runtime object *object* to facilitate mobility, or (3) an application-level message transmitted from one object to another, which for ABS is either a method invocation (Call message) or the resolved value of a future (FUTURE message). Given an application-level message, the function $\texttt{dest}(msg)$ returns the identifier of the intended recipient object, while the function `id` returns the identifier of a given runtime object.

The nature of a FIFO queue Q of messages is specified through three functions: `enqueue`, `dequeue` and `first`. $\texttt{enqueue}(Q, msg)$ returns the queue that results when the message msg is added to the back of Q. If Q is non-empty, $\texttt{first}(Q)$ returns the message at the front of Q, and $\texttt{dequeue}(Q)$ returns the queue that results when the front message is removed. For brevity, $\texttt{enqueue}(Q, msg) = Q'$ is defined as a relation $Q \xrightarrow{\texttt{enqueue}\,(msg)} Q'$, while the conjunction that $\texttt{first}(Q) = msg$ and $\texttt{dequeue}(Q) = Q'$ is defined as $Q \xrightarrow{\texttt{dequeue}\,(msg)} Q'$.

The nature of a routing table is specified through the functions `update`, `next`, `register` and `replace`, and an infix operator $\in$. The function `update` takes three arguments: the routing table τ of the current node, the node identifier u' of the adjacent node, and the routing table τ' of the adjacent node. The function returns a routing table τ'', which incorporates the routes from τ' into τ if appropriate, with the constraint that all such routes must go through the node u'. For brevity, $\texttt{update}(\tau, u', \tau') = \tau''$ is defined as a relation $\tau \xrightarrow{\texttt{update}\,(\tau', u')} \tau''$. The function `next` takes three arguments: the routing table τ of the current node, the object identifier o' of the node we want the next hop for, and the default hop u, which is the identifier of the current node. The function returns the node identifier u' which is the next hop of o' according to the table. The function `register` takes four arguments: the routing table τ of the current node, the object identifier o' of the object we want to add a route for, the node identifier u of a neighbor node (usually self) which is the next hop, and a non-negative integer k for the distance to the object (in all instances in the rules, it is 0). The function returns a routing table τ' which incorporates the new route. For brevity, $\texttt{register}(\tau, o', u, k) = \tau'$ is defined as a relation $\tau \xrightarrow{\texttt{register}\,(o', u, k)} \tau'$. The function `replace` takes four arguments (of the same type as `register`): the routing table τ of the current node, the object identifier o' of the object we want to replace the route for, the node identifier u of a neighbor node which is the next hop, and a natural number k for the distance to the object. The function returns a routing table τ' which has removed any existing routes for o' and added the route given. For brevity, $\texttt{replace}(\tau, o', u, k) = \tau'$ is defined as a relation $\tau \xrightarrow{\texttt{replace}\,(o, u, k)} \tau'$. The claim $o \in \tau$, with a node identifier u given by the context, means that, according to τ, the object with identifier o is located on the node u.

The two parts of the global state can evolve jointly by performing synchronized labeled transitions, but also separately without exchanging information. The rules for such synchronization and separate evolution are shown in Figure 4.2. A label α is either $\texttt{mv}(object)$ (moving an object), $\texttt{rg}(o, o')$ (registering a new object identifier), or $\texttt{tr}(o, msg)$ (transporting a message). Intuitively, a label with an overline means that information is outgoing or being sent, while a label without overline means information is incoming or being received. An ABS-NET execution of an ABS program is a possibly infinite sequence of global states, such that the transition between a previous state and the next is valid. The program is not explicitly represented in a state, since it assumed to always be available unaltered at all nodes. The network topology is static during an execution, and we do not consider failures such as message losses.

$$\text{(Net-Red)}\ \frac{\{net\} \to \{net'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn\}} \qquad \text{(Cn-Red)}\ \frac{\{cn\} \to \{cn'\}}{\{net\}\,\{cn\} \to \{net\}\,\{cn'\}}$$

$$\text{(Cn-Out-Net-In-Red)}\ \frac{\{net\} \xrightarrow{\alpha} \{net'\} \quad \{cn\} \xrightarrow{\overline{\alpha}} \{cn'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn'\}} \qquad \text{(Net-Out-Cn-In-Red)}\ \frac{\{net\} \xrightarrow{\overline{\alpha}} \{net'\} \quad \{cn\} \xrightarrow{\alpha} \{cn'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn'\}}$$

Figure 4.2: ABS-NET reduction rules connecting objects and networks

Intuitively, transitions by the rule Net-Red are driven by node-related events, e.g., timeouts triggering routing table exchanges, or application-level messages being received and routed further. Transitions by Cn-Red are triggered when objects execute ABS program statements that only change internal state. Cn-Out-Net-In-Red is used when program execution requires interaction with the environment to proceed (e.g., method calls) and for object migration. Net-Out-Cn-In-Red is used when a node transmits objects or application-level messages received through links to the local interpreter layer.

4.3.2 Node Controller Behavior

The reduction relation for networks, alluded to Figure 4.2, is defined by the rules in Figure 4.3. The rules apply to subsets of nodes and arcs, such that the elements can be rearranged to match the left-hand side. The labeled transition rules Net-Msg-Recv-Out, Net-Msg-Send-In, Net-Object-Send-In, and Net-New-Object-In, in which a node exchanges information with an object, all use the premise $o \in \tau$ to restrict actions to pertain to node-local objects. This is how object location is reflected in ABS-NET. `fresh`(o) means that the identifier o is globally unique.

For proper progress in execution, we assume networks are such that (1) there are no dangling arcs referencing non-existent nodes, (2) for every arc between nodes there is an arc in the opposite direction, and (3) every node comes with a self-loop arc, i.e., an arc going from and to the node. Self-loop arcs are important for two reasons. First, it allows us to use the same rules for message passing in both the case where the sender object is at a different node from the receiver object, and where the sender is at the same node as the receiver. Once a message has been put in the self-loop queue, it appears as if it came from some other node, and the rule Net-Msg-Recv-Out can be applied. Second, it is not always the case that there is a route to the recipient of a message, because of the possibility of stale routing tables. However, messages must be dealt with somehow, in particular if they are

(Net-Table-Send)
$$\frac{\begin{array}{l} u' \neq u \\ Q \xrightarrow{\mathtt{enqueue}\,(\textsc{Table}\,(\tau))} Q' \end{array}}{\begin{array}{c} \mathsf{nd}\,(u,\tau)\ \mathsf{ar}\,(u,Q,u') \\ \to \mathsf{nd}\,(u,\tau)\ \mathsf{ar}\,(u,Q',u') \end{array}}$$

(Net-Table-Recv)
$$\frac{\begin{array}{l} Q \xrightarrow{\mathtt{dequeue}\,(\textsc{Table}\,(\tau'))} Q' \\ \tau \xrightarrow{\mathtt{update}\,(\tau',u')} \tau'' \end{array}}{\begin{array}{c} \mathsf{ar}\,(u',Q,u)\ \mathsf{nd}\,(u,\tau) \\ \to \mathsf{ar}\,(u',Q',u)\ \mathsf{nd}\,(u,\tau'') \end{array}}$$

(Net-Msg-Recv-Out)
$$\frac{\begin{array}{l} Q \xrightarrow{\mathtt{dequeue}\,(msg)} Q' \\ \mathtt{dest}\,(msg) = o \quad o \in \tau \end{array}}{\begin{array}{c} \mathsf{ar}\,(u',Q,u)\ \mathsf{nd}\,(u,\tau) \\ \overset{\overline{\mathtt{tr}\,(o,msg)}}{\to} \mathsf{ar}\,(u',Q',u)\ \mathsf{nd}\,(u,\tau) \end{array}}$$

(Net-Msg-Send-In)
$$\frac{\begin{array}{l} o \in \tau \quad \mathtt{dest}\,(msg) = o' \\ \mathtt{next}\,(\tau,o',u) = u' \\ Q \xrightarrow{\mathtt{enqueue}\,(msg)} Q' \end{array}}{\begin{array}{c} \mathsf{nd}\,(u,\tau)\ \mathsf{ar}\,(u,Q,u') \\ \overset{\mathtt{tr}\,(o,msg)}{\to} \mathsf{nd}\,(u,\tau)\ \mathsf{ar}\,(u,Q',u') \end{array}}$$

(Net-Route-Further)
$$\frac{\begin{array}{l} Q_1 \xrightarrow{\mathtt{dequeue}\,(msg)} Q_1' \quad \mathtt{dest}\,(msg) = o \quad o \notin \tau \\ \mathtt{next}\,(\tau,o,u) = u'' \quad Q_2 \xrightarrow{\mathtt{enqueue}\,(msg)} Q_2' \end{array}}{\begin{array}{c} \mathsf{ar}\,(u',Q_1,u)\ \mathsf{nd}\,(u,\tau)\ \mathsf{ar}\,(u,Q_2,u'') \\ \to \mathsf{ar}\,(u',Q_1',u)\ \mathsf{nd}\,(u,\tau)\ \mathsf{ar}\,(u,Q_2',u'') \end{array}}$$

(Net-Object-Send-In)
$$\frac{\begin{array}{l} o \in \tau \quad u' \neq u \\ \tau \xrightarrow{\mathtt{replace}\,(o,u',1)} \tau' \\ Q \xrightarrow{\mathtt{enqueue}\,(\textsc{Object}\,(object))} Q' \end{array}}{\begin{array}{c} \mathsf{nd}\,(u,\tau)\ \mathsf{ar}\,(u,Q,u') \\ \overset{\mathtt{mv}\,(object)}{\to} \mathsf{nd}\,(u,\tau')\ \mathsf{ar}\,(u,Q',u') \end{array}}$$

(Net-Object-Recv-Out)
$$\frac{\begin{array}{l} \mathtt{id}\,(object) = o \\ Q \xrightarrow{\mathtt{dequeue}\,(\textsc{Object}\,(object))} Q' \\ \tau \xrightarrow{\mathtt{replace}\,(o,u,0)} \tau' \end{array}}{\begin{array}{c} \mathsf{ar}\,(u',Q,u)\ \mathsf{nd}\,(u,\tau) \\ \overset{\overline{\mathtt{mv}\,(object)}}{\to} \mathsf{ar}\,(u',Q',u)\ \mathsf{nd}\,(u,\tau') \end{array}}$$

(Net-New-Object-In)
$$\frac{\mathtt{fresh}\,(o') \quad o \in \tau \quad \tau \xrightarrow{\mathtt{register}\,(o',u,0)} \tau'}{\mathsf{nd}\,(u,\tau) \overset{\mathtt{rg}\,(o,o')}{\to} \mathsf{nd}\,(u,\tau')}$$

Figure 4.3: Node controller reduction rules

coming from some other node, from which there could be other important messages pending. Hence, they are put in the self-loop queue, i.e., the default next hop of an application-level message is the node itself.

The ABS-NET reduction rules for objects at runtime are deferred to Appendix B. Compared to Core ABS, the state of an object has been extended with an input and an output queues for asynchronous transfer of application-level messages, and a structure to keep track of resolved future values. In contrast, the standard Core ABS semantics handles resolution and querying of futures in a centralized way. In fact, all rule premises from the standard semantics that pertain to more than object-local state are absent in ABS-NET—its decentralized nature is syntactically apparent.

4.4 Adaptation

We consider three QoS objectives against which runtime adaptation solutions can be assessed: node load, arc load and message latency. In our setting, the definition of node load is simple but coarse grained: the load on a node u is the number of objects located on u with active tasks. One advantage of this measure is that it is an intrinsic property of runtime configurations. We need a model-intrinsic measure of load to enable reasoning at an abstract level about convergence to balanced allocation and that loads stay within a certain range. One disadvantage of the approach is that it fails to take into account the varying use of memory and processing power among tasks. However, in an implementation, a more fine-grained measure of load can be adopted, as long as it is linear in the number of active tasks.

We define the load of a particular arc as the number of messages traversing it per simulated unit of time. Hence, global minimization of arc load means that a minimal number of inter-node messages are sent overall, with respect to the current state of routing tables at nodes. Unless all routing tables are optimal (minimum stretch), however, there is no guarantee that the number of hops, i.e., latency, of a particular object-addressed message is minimal.

In our evaluation of runtime adaptation, we use ABS programs that are nonterminating and cyclical. The motivation is that for adaptations to current conditions to have a chance of conveying benefits, similar conditions must hold in the future. There is no obvious payoff in attempting to adapt when future states are random independently of the current state.

Although we wish to simultaneously meet all of our QoS objectives fully, we consider node load balancing our primary concern. Load balancing solutions are also relatively well-studied in the literature, making it easier to find a good starting point. Azar et al. [15] consider the problem of achieving balanced allocations in the framework of stochastic processes, where it is viewed as stepwise allocations of balls into bins. They highlight the use of greedy schemes for quickly converging to a ball-to-bin assignment where the maximum number of balls in any bin is minimized. The main drawback of this approach in a distributed setting is the reliance on atomic, single assignments of a ball to a bin at each algorithm step. Even-Dar and Mansour [68] study load balancing in a distributed setting where allocations are not necessarily done one-at-a-time. They give a distributed algorithm for selfish rerouting that quickly converges to a Nash equilibrium, which corresponds to a balanced resource allocation. However, at each round, locally computing a new allocation requires having exact knowledge of all loads in the system, which is complicated and costly to acquire in the current setting. Berenbrink et al. [17] describe and analyze fully distributed algorithms which require only local knowledge of the total number of resources and the load of one other resource to perform a single task migration step. The algorithms, some of which have attractive expected time for convergence, can be straightforwardly translated to a synchronous, round-based distributed setting and further to a message-passing setting, assuming some inherent synchrony. One important assumption made in the algorithm analysis is that a task can migrate to any other resource in a single concurrent round. For this property to hold, the underlying network graph must be complete, which we do not generally assume.

A factor in the convergence time is whether neutral moves are allowed, i.e., whether a migration can happen even when, as far as can be told locally, the move does not result in a more balanced allocation but merely an equally good one. For allocations in a sparse network graph where load differences between neighbors are one, there can nevertheless be maximal load differences in the order of the graph diameter, which can be significant. With neutral moves, such allocations can be improved on.

The problem of oscillating behavior during task balancing can be mitigated by the use of coin flips before finalizing decisions to migrate tasks, as in the algorithms of Berenbrink et al. Oscillation can be worsened by reliance on stale information, but if the information is not *too* stale, oscillation periods can sometimes be bounded [71].

The literature on load balancing related to scientific computing contains work on simultaneously optimizing task allocations and communication overhead. For example, Cosenza et al. [48] give a distributed load balancing scheme for simulations involving agents moving in space from worker to worker. The scheme, which is validated experimentally, optimizes both worker load and communication overhead between workers, but assumes only a small area of interest for each agent, with agents unable to communicate with other agents outside this area. In the current work, two objects can communicate whenever the identifier of one of them is known to the other, making it harder to minimize communication overhead. Catalyurek et al. [34] describe how to use hypergraph partitioning to minimize both communication volume and migration time of tasks for parallel scientific computations. However, the repartitioning is performed in batch and requires complete knowledge of the data and computations on each node.

At this initial stage of the work, we do not consider the cost of migration itself in terms of messaging and other resources. Hence, we only measure communication in terms of messages exchanged between objects, ignoring overhead in terms of routing and load-related messages.

4.5 Simulator

We have evaluated our runtime adaptation approach by developing a simulator for running ABS programs in a network of nodes according to the ABS-NET semantics. We have run the simulator with a variety of network node topologies and object migration procedures on a number of proposed synthetic scenarios defined by ABS programs.

Our simulator's main purposes are to serve as a proof-of-concept for ABS-NET and to allow us to run adaptability case studies with particular programs and topologies. Specifically, we are interested in studying convergence properties of object migration procedures in practice, and in showing that our approach to distributed execution scales to networks with many nodes.

The simulator is implemented in Java. Each node controller is implemented as a Java thread, which communicates with other controllers through TCP sockets, using the KryoNet network library [112]. One reason for the choice of sockets is to enable to scale simulations over several physical machines and a large number of simulated network nodes. All node controllers in the network have a representation of the abstract syntax tree of the ABS program being executed, which is generated from ABS program code by the lexing and parsing frontend shared by most ABS backends.

As in the conceptual model and the formal semantics, a node controller can have zero or more objects, each having at most one active task. An active task has a reference to the statement currently being executed in the abstract syntax tree. We call an object active if it has an active task. Scheduling of active tasks is done at the node controller level in a round-robin fashion for active objects. More precisely, the scheduler deterministically steps all active tasks, checks for active objects, and then repeats the process on the new set of active tasks.

We implement statement execution by interpretation. The main reason for this choice is to enable easy serialization of objects between executing statements; to get immediate results from load balancing, we must be able to migrate active objects. One drawback of using interpretation is that local execution is slow and resource-demanding compared to execution in the standard ABS backends.

A node controller is associated with a unique TCP port on the host system. Besides a list of neighbor handles, which abstract over underlying sockets, and a list of local objects, the node controller maintains a routing table. The routing table is broadcasted to neighbors after entries have been changed or added as a result of statement execution or incorporation of routes from neighbor messages. Hence, except when many locations have been updated in a short interval, we expect routing tables to be up-to-date or nearly so, taking into account the network size restrictions of the simulator.

Network topology setup and program loading is handled by scripting on top of a custom simple command-line interface (CLI). When starting up, a node controller is assigned a migration procedure through the CLI, which is the same for all node controllers in the network. One desirable feature that is not implemented is CLI control of link characteristics, such as delays.

By default, the simulator starts the initial task on a single startup node. The initial task is defined by the statements in the mandatory starting block of the ABS program. In all our programs, this task creates all the objects used for the duration of the program. Migration and logging does not commence until a method with the name `setupFinished` is called on some object. There are several reasons for this kind of initialization: it is easier to predict load balancing behavior with a fixed set of objects, and it is problematic to create new objects on the fly without garbage collection, which we have not implemented.

4.5.1 Scenarios

A network configuration determines the size and topology of the network; large and dense networks give more overhead in the form of routing and load messaging, making simulations slower. Currently, highly connected topologies with in the order of 25 network nodes can be simulated in reasonable time. On this note, we limit the evaluation to networks with three distinct underlying network topologies from sparsely to fully connected: grids, hypergraphs and full meshes. Our base initial setup for each topology has 32 nodes. Since the simulator scales to at least in the order of 100 nodes for sparsely connected topologies, we also investigate grids larger than 32 nodes to compare results.

For defining object behavior, we have developed a number of ABS programs specifically to run in our simulator. All programs have a setup phase, where a fixed number of objects are initialized, and a phase where the generated objects perform some computation, possibly involving communication; there are no short-lived dynamically created objects. For all programs but one, which implements the Chord distributed hash table (DHT) algorithm [190], communication patterns among generated objects follow straightforwardly from the code. This makes it easier to follow what happens during a simulation and to reason about how far an allocation of objects to nodes is from the optimum, factors which we considered particularly important in scenario development. After running initial simulations, we have adjusted parameters in our programs, and in some cases added functionally redundant instructions to get constant and consistent load and messaging, since our migration procedures consider mainly objects with active tasks. With spurious activity among nodes, messaging and load varies greatly, and progress becomes hard to discern. The programs below are available online [158]:

`IndependentTasks.abs` The starting task generates objects, and each generated object is called upon to perform a long-running task. There is no communication among workers—only briefly at startup between the coordinator object, which initializes and assigns tasks, and the generated objects. Since there is no communication, an optimal allocation is an even distribution of objects among nodes, regardless of the network topology.

`Star.abs` An object star configuration consists of one center object and one or more fringe objects. The fringe objects in the star continually communicate with the center object, but not among themselves. The program builds a number of independent object star configurations.

`Ring.abs` The starting task generates objects which know the identifiers of the next object in the ring. The last object generated gets the identifier of the first object. The first object, when called, calls its next object, and so on, until the object which has the first object as next object is reached. In the computation phase, many such calls traverse the ring simultaneously.

`ChordDHT.abs` An implementation of the Chord DHT algorithm. Key-value mappings are distributed between a number of objects, which all support a put/get interface to clients. Objects are arranged in a ring, but aside from references to their neighbors, each object has $\log(n)$ “fingers”, references to non-adjacent objects, where n is the size of the keyspace. The addition, or join, of an object to the ring places the new object at a particular position based on its identifier and can trigger global reconfiguration of the ring. During setup, 128 objects are joined to the chord, and each object becomes associated with either a producer object, which continually puts values into the DHT, or a consumer object, which continually attempts to retrieve values from the DHT using pseudorandom keys.

We consider only migration procedures that as a first priority balance out load evenly among nodes in the network. As a consequence, a simulated node controller continually informs neighbor nodes of its load when appropriate, and receives load messages from neighbors in turn, regardless of the migration procedure used. In the simulator, each migration procedure defines a callback method which takes the affected node controller as a parameter. The callback method is invoked, and can result in the migration of several objects to neighbor nodes. The migration procedures used are described below.

Berenbrink et al. An adapted version of the distributed load balancing algorithm by Berenbrink et al. [17], which does not allow neutral moves. One notable difference in the simulator implementation from the abstract description given in Algorithm 1 is that only a fixed small number of objects (20) have the possibility to migrate in each cycle, because of limits on the sizes of message buffers.

Berenbrink et al. with neutral moves An adapted version of the distributed load balancing algorithm by Berenbrink et al., which does allow neutral moves, and therefore converges more slowly. The only difference from Algorithm 1 is that the if-condition is $l > l'$ instead of $l > l' + 1$. As determined experimentally, only migrating one or two objects per node per cycle leads to significantly less oscillation of objects, compared to when migrating three or more.

Berenbrink et al. with communication intensity A variant of the preceding procedure, where objects are selected for migration based on their affinity to the (randomly) chosen neighbor node, as determined by their communication history with objects in the neighbor node's direction. The communication history is a list of other objects that a given object has communicated with recently, as given by abstract object-local time, defined by the number of tasks finished since initialization. The affinity of an object to the neighbor node is then quantified as the number of objects in the communication history that are located in the direction of the node, according to the routing table.

Weighted neighbor load difference Once every cycle, an object and an adjacent node are chosen uniformly at random and independently. Then, a probability of migration is calculated and enacted based on the difference in load between the current node and the chosen node, with probability 1 for a difference of 10 or more, and probability 0 for a negative difference. If the load difference is d, the migration probability becomes $\frac{d}{10}$, adjusted to closest number in the interval $[0, 1]$.

Weighted neighbor load difference with communication intensity Given a randomly chosen object and adjacent node as in the previous procedure, we define the probability of migration according to communication intensity as the number of entries in the object's communication history found in the direction of the node, divided by the total number of entries in the history. This probability is then combined via weighted averaging with the neighbor load difference probability to define the weighted neighbor load with communication procedure. We have used the weight 0.2 for the communication intensity probability and 0.8 for the neighbor's load probability.

Algorithm 1 Berenbrink et al. load balancing cycle

```
for each active object $o$ do
    let $u'$ be a neighbor chosen uniformly at random
    let $l$ be the current load, let $l'$ be the last known load of $u'$
    if $l > l' + 1$ then send $o$ to $u'$ with probability $1 - l'/l$
```

4.5.2 Scenario Objectives

Since our primary objective is to balance node load evenly, we record the load of all individual nodes over time, and then compute the maximum load and load standard deviation. For scenarios with little to no object communication, these are the only measures that are relevant with respect to our objectives. For scenarios with significant messaging, we also consider the number of object-related messages sent (i.e., CALL and FUTURE messages) by each node between sampling intervals—with the average number of messages and standard deviation shown. We do not count messages sent by a node to itself via the self-loop arc, since such messages need not go through a physical link in an implementation.

We sample the required quantities from simulations at a fixed global rate, corresponding roughly to a certain number of transitions (1000) in the semantics with imposed fairness via round-robin scheduling.

4.5.3 Results

In this section, we describe simulation results for the scenarios given above.

Simulations of `IndependentTasks.abs`

The program creates 201 objects in total: one starting object which becomes inactive after initialization and 200 objects that each have a task that runs for the course of the program.

As expected, the algorithm by Berenbrink et al. without neutral moves converges very quickly and stays unchanged with no migrations after reaching a state where neighbor load differences are at most one, which on a full mesh is always balanced. For most of the runs on a 32-node hypergraph network topology, the stable state coincided with a completely balanced allocation, or very closely so. For the grid case, the stable allocation in almost all cases deviated significantly from a fully balanced one.

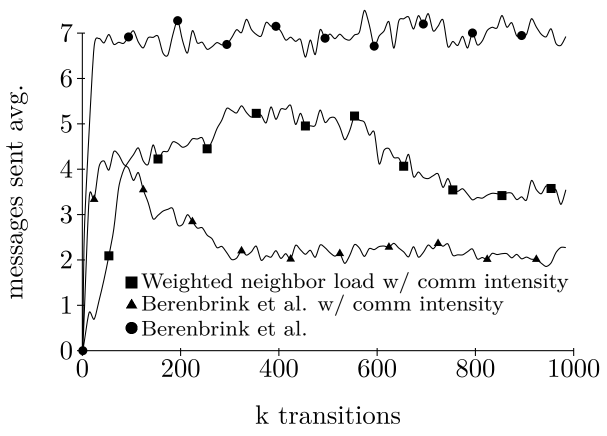
The algorithm variant with neutral moves and two migrations per cycle converges to an almost-stable state quite quickly on a hypergraph, but continues to have minor oscillation of objects. With the same algorithm and five object migrations allowed per cycle, there is considerably more oscillation going on after coming close to a balanced allocation. On a grid topology, where a stable allocation can be further away from a balanced allocation, allowing neutral moves gives better results than disallowing them, as expected. For a grid, the gain from using neutral moves is most distinct in a lower standard deviation compared to the algorithm without neutral moves.

Simulations of `Star.abs`

In the star program, stars are constructed so that each node can hold a whole star, and there is precisely one star per network node. In an optimal allocation, therefore, there are no node-to-node message exchanges at all; all messages are sent locally.

We expected the pure load balancing procedures to have markedly worse results than the procedures taking inter-object communication intensity into account. The average number of sent messages and the standard deviation of sent messages over time for the star program on a grid is shown in Figure 4.4, with measurements smoothed out via averaging over five samples to reduce noise. As can be seen, there is a distinct improvement with respect to messages sent when using the algorithm by Berenbrink et al. augmented with message intensity comparisons when compared to the other procedures, although it is quite far from the optimum. The algorithm using probabilistic weighting of load and messaging seems to improve the most over time, although it performs similarly to the messaging-augmented load balancing algorithm by Berenbrink et al.

With all the tested migration strategies for a grid, load became evenly balanced relatively quickly, as seen in the upper part of Figure 4.5. Hence, there was no significant avoidance of messaging by communicating objects clustering at a few specific nodes.



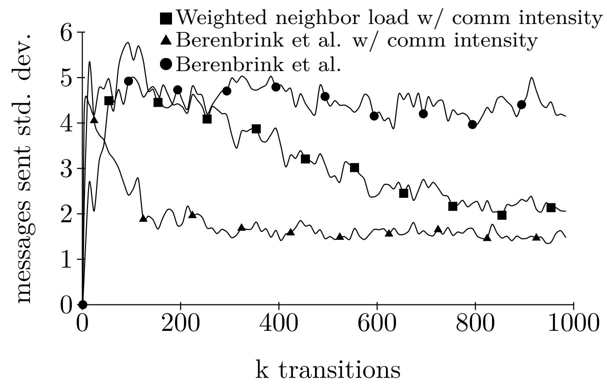
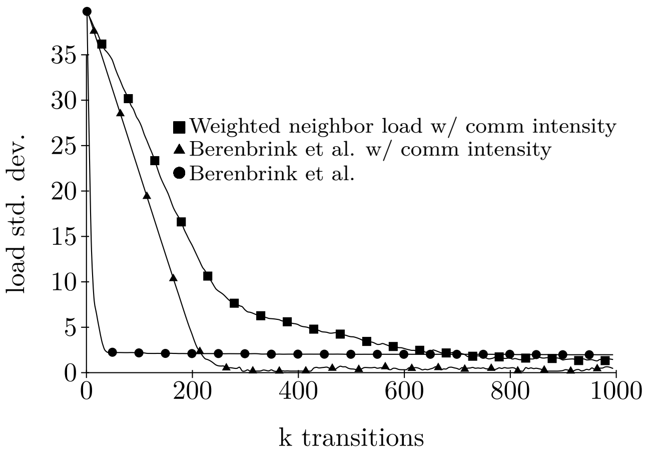


Figure 4.4: `Star.abs` on a 32-node grid, messages sent

Because of the simplicity of the object communication graph and the fact that it is possible to reach an allocation where no inter-node communication takes place, it is worthwhile to illustrate how near specific algorithms can get after many (1000) cycles, for comparison. In a given allocation, each object has a total distance in hops to the other object it communicates with. For fringe objects, the total distance is the number of hops to its center object, but center objects have total distance equal to the sum of all distances to its fringes. In an optimal allocation, all centers (and all fringes) have total distance zero. In the lower part of Figure 4.5, gray bars show the distribution of total distance among the 32 center objects on a grid for the load balancing algorithm by Berenbrink et al. The black bars show the distribution of total distances of the objects for the algorithm by Berenbrink et al. augmented with message intensity comparisons. The distributions intersect, but the former



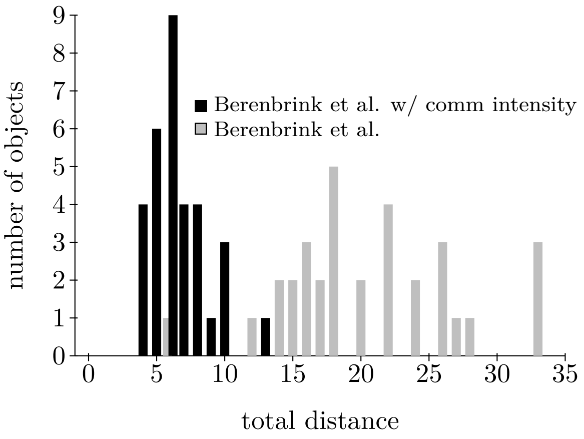


Figure 4.5: `Star.abs` on a 32-node grid, load std. dev. and object distances

algorithm fares worse.

Results for `Star.abs` on a hypergraph topology give a less pronounced advantage to the two migration procedures which take message intensity into account. Of those procedures, the Berenbrink et al. variant produces the least messaging, but trends are largely the same as for the grid case; hence, we omit plots. For the case of a complete topology, the amount of messaging was virtually the same for all procedures. An intuition for why this is the case is that it becomes much harder to improve upon an allocation in a situation where migrations are helpful only when communicating objects end up on the same node, and there is additionally no corresponding loss of proximity to another object. Grids of 64 and 128 nodes have the same messaging trends as the 32-node case.

Simulations of `Ring.abs`

When running a ring of 128 objects on a 32-node grid, there are balanced allocations with all nodes having 4 objects, where all objects that communicate are either on the same node or adjacent nodes. The idea is that two of the objects on a node are part of a segment of the ring, while the other two are part of another segment coming back the other way. In such allocations, at most one inter-node message per object is needed for a method invocation that involves the whole ring.

Figure 4.6 shows the average number of messages sent of a 128-object ring on a grid topology, while the upper right half shows the standard deviation of the number of sent messages; smoothing by averaging samples has been applied in both cases. The pattern from the star program remains, with procedures taking messaging into account performing better, but the differences are smaller. The progressively decreasing number of inter-node messages sent are not due to clustering of many objects on a few nodes, as shown by the eventually low standard deviation of load in the lower left part of the figure.

In the lower part of Figure 4.7, gray bars show the distribution of total distance among all ring objects on a grid to the objects they communicate with, after 1000 migration cycles using the algorithm by Berenbrink et al. Black bars show the distribution for the algorithm by Berenbrink et al. with neutral moves augmented with message intensity comparisons. There is overlap, but the latter algorithm results in many more objects with total distance between 1 and 5. However, both distributions are quite far from being optimal.

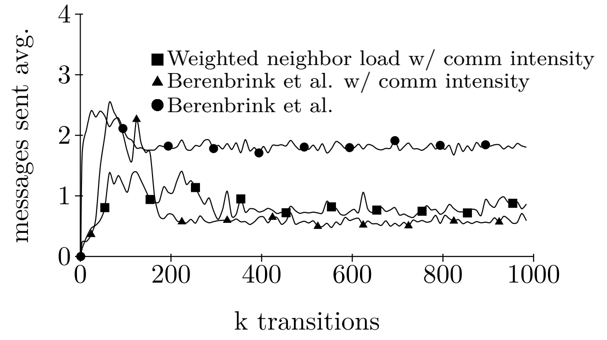
As in the case of `Star.abs`, the performance trend in messaging over time is largely the same on a grid and hypergraph topology for `Ring.abs`. The main difference on a hypergraph is that procedures which take message intensity into account result in less pronounced improvements over the pure load balancing procedure. For a complete topology, differences are once again small, but with an edge towards the message intensity procedures. Once more, grids of 64 and 128 nodes preserve the trend from the 32-node case.

Simulations of `ChordDHT.abs`

In the Chord DHT program, the weighted neighbor's load and message intensity strategy exhibited a tendency to quickly cause message buffer overflows, while the procedures based on the algorithms of Berenbrink et al. worked largely as expected.

The upper part of Figure 4.8 shows the average number of messages sent for nodes when running the program on a grid, and the lower part shows the standard deviation of the number of messages. Again, smoothing by averaging samples five at a time has been applied. The results suggest that there is a reasonable payoff from taking messaging into account in a migration strategy, even when running a program with relatively complex communication patterns.

Simulations of `ChordDHT.abs` on a hypergraph show very similar trends in performance to the grid case, but give a less pronounced advantage to procedures



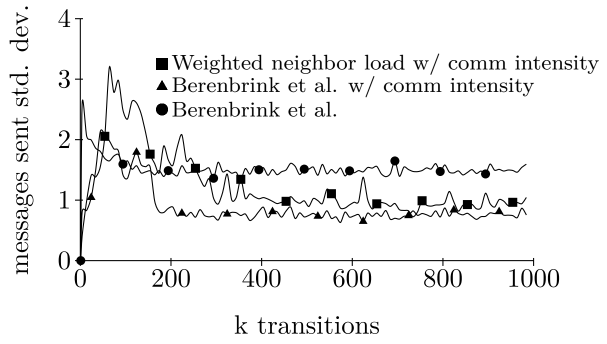
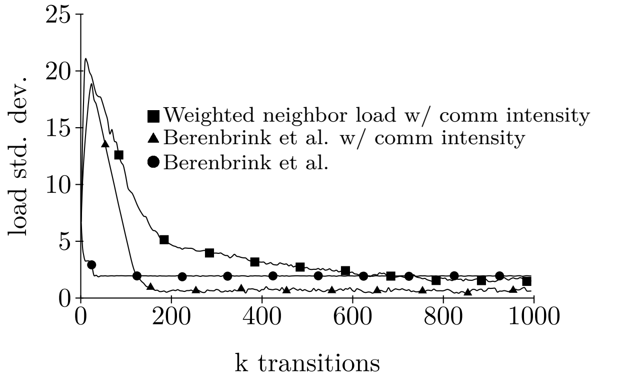


Figure 4.6: `Ring.abs` on 32-node grid, sent messages

which take message intensity into account, as for previous programs. In a fully connected topology, the procedures result in effectively the same amount of messaging, as before.

4.6 Conclusions and Future Work

The simulation results suggest that it is feasible in a decentralized setting to meet the objective of balanced resource allocation, and also make headway towards the objective of minimizing communication of distributed objects. The results also validate the applicability of the ABS-NET model with location-independent routing to decentralized runtime adaptation. The main concern for relevance to real-world networks is the use in the model of unbounded message queues, and the lack of rate limitation and latency controls in our simulator. In future work, we plan to continue the theoretical and simulation-based studies to deepen our understanding of multi-dimensional resource management, improve the performance and accuracy of the simulator, and investigate adaptation in dynamic networks, initially only with benign churn, i.e., with controlled startup and shutdown of nodes.



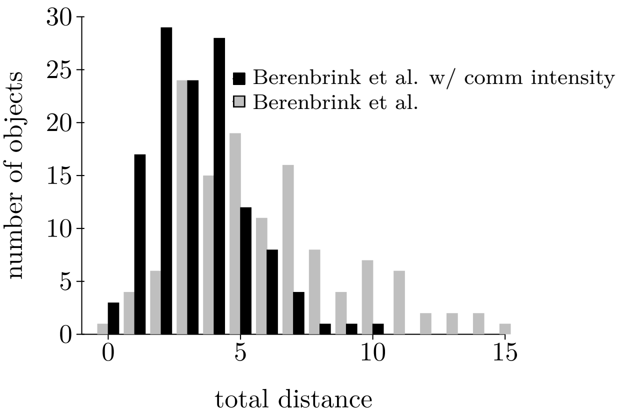
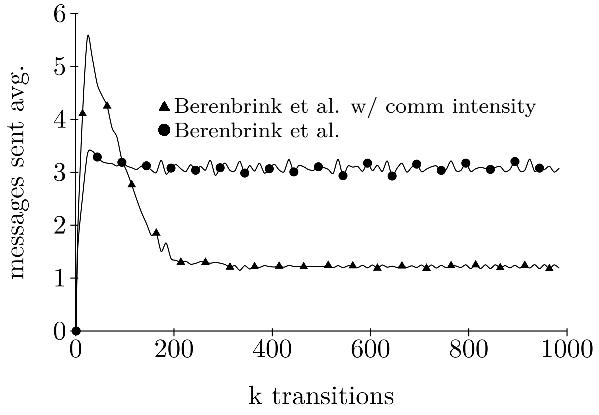


Figure 4.7: `Ring.abs` on 32-node grid, load std. dev. and object distances

Acknowledgements

We thank the anonymous reviewers for their comments and suggestions, which were of significant help in improving the paper. We also thank our colleagues in the HATS project for useful discussions and criticism.



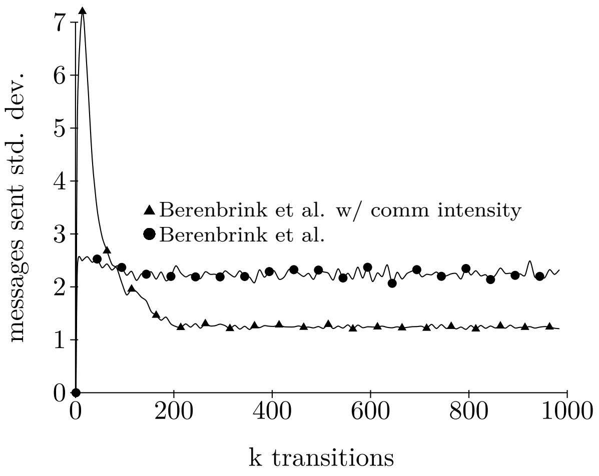


Figure 4.8: `ChordDHT.abs` on 32-node grid

Chapter 5

Decentralized Adaptive Power Control for Process Networks

Mads Dam Karl Palmskog

KTH Royal Institute of Technology, Sweden
{mfd, palmskog}@kth.se

Abstract

Mobility and location transparency of distributed objects enable efficient resource allocation in networks, and can be effectively realized at the language level using location independent routing. However, the benefits of mobility are restricted by the available resources, e.g., in the form of processing nodes, which may be too few or too many to suit an application. To continually meet an application developer's requirements on computational task throughput, and an infrastructure provider's requirements on energy usage, the network itself must be able to adapt. In this paper, we consider fault-free networks of nodes connected point-to-point by asynchronous message passing channels, and propose a protocol for shutdown of nodes that preserves the integrity of distributed objects. This protocol enables decentralized power control, where nodes are turned on and off in response to computational requirements. We analyze the protocol both by verifying a restricted version in the model checker Spin, and by formulating a transition system model and proving properties by induction in that model for networks of arbitrary size. We define a protocol extension that, while using only a node's neighborhood-local information, is sufficient to ensure networks remain connected after node shutdown, and outline more complex, general local criteria. Finally, we discuss heuristics for node-local decision making on initiating a shutdown process, to meet adaptability requirements.

5.1 Introduction

For distributed object programs executing in large networks, *location transparency* is an important property, ensuring that inter-object messages are delivered to their destinations—regardless of the location of the sender och receiver. To enable efficient resource allocation, objects must also be *mobile* across nodes, which turns location transparency into an even greater challenge.

In previous work, described in Chapter 2 and Chapter 3, we use location independent routing to achieve location transparency of mobile distributed objects executing on nodes connected point-to-point through asynchronous message passing channels. However, we there rely on an assumption that the abstract network of processing nodes is static over the course of a program execution. In contrast, real networks are dynamic in at least two different ways. First, nodes can crash, recover, and deviate from arbitrarily from expected behavior. Second, nodes can be added to a network or be shut down in a controlled way, according to requirements on, e.g., throughput, computational load, or energy consumption. In this paper, we concentrate on the second case of network dynamicity, by investigating decentralized node shutdown in asynchronous networks that preserves the integrity of objects and inter-object messages located on nodes or passing through links.

We propose a maximally nondeterministic protocol for node shutdown, reminiscent of a two-phase commit protocol [187], and provide evidence of its correctness with respect to the integrity of an executing distributed program. Part of the evidence is in the form of a verified abstract model of the protocol in the language Promela of the model checker Spin [95]. Since this bounded model does not account for behavior in networks of arbitrary size, we formulate a transition system model in an inductive framework, and use rule induction to prove safety properties of valid network configurations. However, this analysis falls short of full protocol verification, which also includes liveness in the form of eventual node shutdown. We also describe how to maintain a connected network, to prevent scenarios where program execution is unable to progress due to objects becoming unreachable. Finally, we discuss strategies, enabled by the node shutdown protocol, for decentralized right-sizing of process networks according to requirements on load and energy consumption.

The paper is organized as follows. In Section 5.2, we describe the behavior of distributed objects, and in Section 5.3, we provide a formal definition of the network model, parameterized on object behavior. Section 5.4 introduces the node shutdown protocol, both by example and by semi-formal definition. Section 5.5 describes verification of the protocol through model checking a bounded model in Spin [95] and by induction on the rules of a transition system, using the Coq proof assistant [43]. In Section 5.6 and Section 5.7, we discuss how to maintain a connected network graph and possible strategies for decentralized power control, respectively. We survey related work in Section 5.8 and conclude in Section 5.9.

5.2 Active Object Behavior

We consider collections of distributed objects that execute computational tasks. Each object has its own identity, thread of control, and local data store. Objects can perform local computation and change their state based on the results of computation, and can communicate with each other using asynchronous message passing; no specific processing order of messages is assumed for recipients. Most closely, this view of objects matches the properties of active objects [30] or actors in the Actor model [3]. However, at a coarser level of granularity, objects can also be *virtual machines* [16].

More formally, an object consists of an identifier o and state a, and is written $\mathsf{o}\,(o, a)$. An inter-object message consists of the recipient's identifier, o, and a payload p, and is written $\mathsf{m}\,(o, p)$. From the view of a developer, or programmer, which defines the behavior of objects by writing programs, object task execution abstractly takes place in an evolving configuration of objects and messages. How objects and messages evolve is given by a reduction relation $\xrightarrow{ob}$ on configurations. For example, this reduction relation can correspond to the relation for the network-oblivious semantics of Core ABS in Appendix A.

5.3 Network Model

The network consists of a set of nodes with links in the form of OSI layer 2 interconnects. A node is therefore aware of its neighboring nodes and can communicate with them using asynchronous message passing. Objects are located on nodes and execute tasks locally. Objects can migrate between nodes through transmission of object and task state in messages. While executing object tasks, the network does not necessarily remain static; new nodes can be added, and existing nodes can begin an ordered shutdown process, where all objects are migrated away and all object-related messages routed to neighbors. The only way a message can be lost is if it is in transit from or to a node that shuts down.

The model is intended to capture a scenario where a fixed, or slowly evolving, network infrastructure is used to execute distributed programs, with power control enabling *right-sizing* [102] of the network of active (turned-on) nodes, e.g., to minimize energy consumption. At the same, object migration can be used to achieve *load balancing* of computational tasks, as described in Chapter 4. We assume that each node maintains a routing table and uses location independent routing, as defined in Chapter 2, to forward inter-object messages to their destinations, and therefore periodically exchanges routing information with its neighbors. Consequently, nodes exchange at least three types of messages: routing table messages, object messages, and inter-object (object-addressed) messages.

5.3.1 Node Behavior

Abstractly, a network *net* can be described, through the syntax in Table 5.1, as a configuration of unique nodes and arcs. Objects executing at nodes are equipped with input and output queues for messages, as in previous chapters. Note that collections of objects *os* are considered modulo associativity and commutativity.

Figure 5.1 shows the reduction rules, applied to partial configurations in the rewriting logic style [40], that describe object execution, as permitted by the given reduction relation $\xrightarrow{ob}$ for a network-oblivious semantics. We use the standard FIFO queue operations `first`, `dequeue`, and `enqueue`, with `emp` the empty queue. `register` is used to store information on object location and distance in routing tables. In rule Obj-New, an object o creates a new object o' with some initial state a'' and empty input and output queues, which becomes registered on the node. In rule Obj-Compute, the object o performs some internal computation to reach a new state. Rule Obj-Send captures the process of the object o sending an object-addressed message, which is enqueued in its output queue. Dually, in rule Obj-Receive, the object o evolves after processing a message in its input queue.

Figure 5.2 shows the reduction rules for object mobility and passing object-related messages between nodes. The rules Net-Send-Object and Net-Receive-Object capture mobility of objects between nodes as messages, with accompanying registration in routing tables. The rules Net-Send-Message, Net-Receive-Message, and Net-Route-Message allow distribution of object-related messages, and use the operation `next` to get the identifier of the next hop neighbor to reach a certain object. We omit rules for the cases when routing tables are incomplete, and for exchange of routing table messages between nodes.

o	object identifier	os	$::=$	$\varepsilon \mid os\, os' \mid obj$
p	object message payload	obj	$::=$	$\mathsf{o}\,(o, a, q_{in}, q_{out})$
a	object state	net	$::=$	$\epsilon \mid net\, net' \mid node \mid arc$
u	node identifier	arc	$::=$	$\mathsf{ar}\,(u, q, u')$
q	queue of *msg*	$node$	$::=$	$\mathsf{nd}\,(u, t, \{os\})$
t	routing table	msg	$::=$	$\textsc{Table}(t) \mid \textsc{Object}(obj)$
				$\mid \textsc{Message}(o, p)$

Table 5.1: Network and object configuration syntax

We define a network-aware *execution* as a sequence of network configurations, where each adjacent pair makes a valid transition according to the rules. As shown in Chapter 2 and Chapter 3, it is possible to prove, for a fixed network-oblivious language, that network-aware executions in static networks correspond, through the notion of contextual equivalence, to network-oblivious executions. The problem of benignly dynamic processing networks is to preserve objects and object-related messages when nodes shut down, by suitably restricting the possible execution space.

(Obj-New)

$$\dfrac{\begin{array}{l}\mathsf{o}\,(o,a) \xrightarrow{ob} \mathsf{o}\,(o,a')\,\mathsf{o}\,(o',a'')\\ \texttt{register}\,(o',u,t)=t'\end{array}}{\begin{array}{c}\mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a,q_{in},q_{out})\ os\}) \to\\ \mathsf{nd}\,(u,t',\{\mathsf{o}\,(o,a',q_{in},q_{out})\,\mathsf{o}\,(o',a'',\mathsf{emp},\mathsf{emp})\ os\})\end{array}}$$

(Obj-Compute)

$$\dfrac{\mathsf{o}\,(o,a) \xrightarrow{ob} \mathsf{o}\,(o,a')}{\begin{array}{c}\mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a,q_{in},q_{out})\ os\}) \to\\ \mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a',q_{in},q_{out})\ os\})\end{array}}$$

(Obj-Send)

$$\dfrac{\begin{array}{l}\mathsf{o}\,(o,a) \xrightarrow{ob} \mathsf{o}\,(o,a')\,\mathsf{m}\,(o',p)\\ \texttt{enqueue}\,(\textsc{Message}(o',p),q_{out})=q'_{out}\end{array}}{\begin{array}{c}\mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a,q_{in},q_{out})\ os\}) \to\\ \mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a',q_{in},q'_{out})\ os\})\end{array}}$$

(Obj-Receive)

$$\dfrac{\begin{array}{l}\mathsf{o}\,(o,a)\,\mathsf{m}\,(o,p) \xrightarrow{ob} \mathsf{o}\,(o,a')\\ \texttt{first}\,(q_{in})=\textsc{Message}(o,p)\\ \texttt{dequeue}\,(q_{in})=q'_{in}\end{array}}{\begin{array}{c}\mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a,q_{in},q_{out})\ os\}) \to\\ \mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a',q'_{in},q_{out})\ os\})\end{array}}$$

Figure 5.1: Rules for object execution on nodes

(Net-Send-Object)

$$\dfrac{\begin{array}{l}u \neq u' \quad \texttt{enqueue}\,(\textsc{Object}(obj),q)=q'\\ \texttt{register}\,(\texttt{id}\,(obj),u',t,1)=t'\end{array}}{\begin{array}{c}\mathsf{nd}\,(u,t,\{obj\ os\})\,\mathsf{ar}\,(u,q,u') \to\\ \mathsf{nd}\,(u,t',\{os\})\ \mathsf{ar}\,(u,q',u')\end{array}}$$

(Net-Receive-Object)

$$\dfrac{\begin{array}{l}\texttt{first}\,(q)=\textsc{Object}(obj) \quad \texttt{dequeue}\,(q)=q'\\ \texttt{register}\,(\texttt{id}\,(obj),u,t,0)=t'\end{array}}{\begin{array}{c}\mathsf{ar}\,(u',q,u)\,\mathsf{nd}\,(u,t,\{os\}) \to\\ \mathsf{ar}\,(u',q',u)\,\mathsf{nd}\,(u,t',\{obj\ os\})\end{array}}$$

(Net-Send-Message)

$$\dfrac{\begin{array}{l}\texttt{first}\,(q_{out})=\textsc{Message}(o',p) \quad \texttt{dequeue}\,(q_{out})=q'_{out}\\ \texttt{next}\,(o',t)=u' \quad \texttt{enqueue}\,(\textsc{Message}(o',p),q)=q'\end{array}}{\begin{array}{c}\mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a,q_{in},q_{out})\ os\})\,\mathsf{ar}\,(u,q,u') \to\\ \mathsf{nd}\,(u,t',\{\mathsf{o}\,(o,a,q_{in},q'_{out})\ os\})\ \mathsf{ar}\,(u,q',u')\end{array}}$$

(Net-Receive-Message)

$$\dfrac{\begin{array}{l}\texttt{first}\,(q)=\textsc{Message}(o,p) \quad \texttt{dequeue}\,(q)=q'\\ \texttt{enqueue}\,(\textsc{Message}(o,p),q_{in})=q'_{in}\end{array}}{\begin{array}{c}\mathsf{ar}\,(u',q,u)\,\mathsf{nd}\,(u,t,\{\mathsf{o}\,(o,a,q_{in},q_{out})\ os\}) \to\\ \mathsf{ar}\,(u',q',u)\,\mathsf{nd}\,(u,t',\{\mathsf{o}\,(o,a,q'_{in},q_{out})\ os\})\end{array}}$$

(Net-Route-Message)

$$\dfrac{\begin{array}{l}\texttt{first}\,(q_1)=\textsc{Message}(o,p) \quad \texttt{dequeue}\,(q_1)=q'_1\\ \texttt{next}\,(o,t)=u'' \quad u \neq u'' \quad \texttt{enqueue}\,(\textsc{Message}(o,p),q_2)=q'_2\end{array}}{\begin{array}{c}\mathsf{ar}\,(u',q_1,u)\,\mathsf{nd}\,(u,t,\{os\})\,\mathsf{ar}\,(u,q_2,u'') \to\\ \mathsf{ar}\,(u',q'_1,u)\,\mathsf{nd}\,(u,t,\{os\})\,\mathsf{ar}\,(u,q'_2,u'')\end{array}}$$

Figure 5.2: Rules for object mobility and message passing between nodes

5.4 Shutdown Protocol

The goal is to formulate a protocol for node shutdown, parameterized on a semantics of object behavior, that can be proven correct and be implemented with minimal changes. Informally, the protocol aims to preserve objects and object-related messages when nodes shut down, and ensure progress; formal correctness is deferred to Section 5.5. From the point of view of an implementation, the main omission is node-local scheduling and timing of events.

5.4.1 Node State Machine

Besides having a unique identity, a routing table, and a collection objects, nodes running the shutdown protocol maintain a (high-level) state value, which is either IDLE, TRANSACT, CLEAR, or SHUTDOWN. All nodes that are added to the network have initial state IDLE, which is maintained until the node starts an attempt to shut down by entering the TRANSACT state. While in TRANSACT, the node informs its neighbors that it wants to shut down. If the node receives some negative reply, it enters ABORT, and sends cancellations to all nodes which have been previously notified of the shutdown attempt. If the node receives positive replies from its neighbors, it enters the CLEAR state and starts to transfer away its objects. When all objects are gone, the node enters SHUTDOWN and requests a final confirmation from its neighbors. After all replies have been received, the node finally shuts down and disappears from the network. Figure 5.3 shows the resulting node state machine.

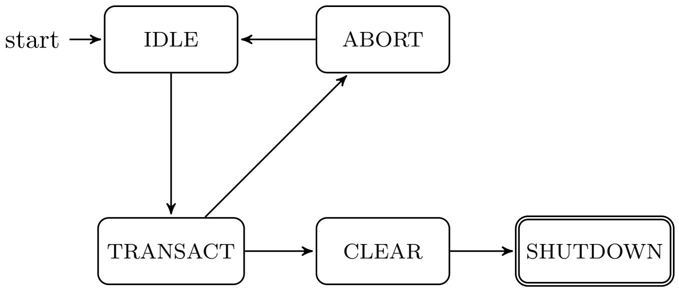


Figure 5.3: Shutdown protocol state machine

5.4.2 Messages

In addition to the messages defined in Table 5.1, the shutdown protocol introduces a number of new message types, as given below.

- Notify is a message sent either to a previously unknown neighbor node, or to a neighbor from which such a message was received. The purpose of this exchange is to prevent transmission of object-addressed and object messages from newly started nodes to nodes that are in the final stages of shutting down.

- Prepare is message sent from a node to its confirmed neighbors when it has started the process of shutting down by entering the state TRANSACT.

- Ready is a message sent in reply to a Prepare message, if the receiver is in state IDLE or is attempting to shut down but loses the tie breaker, as explained below. Whenever Ready is sent to a neighbor, that node becomes blocked.

- Abort is a message sent to unblocked neighbors to which the sender has previously received Ready from. Its main function is to let those nodes unblock the neighbor which failed progress in its shutdown process.

- Shutdown is a message sent from a node that is in the last stage of shutting down. The message is sent to all known neighbors after all local objects have been sent away and the node has entered the SHUTDOWN state. The message's function is as an indicator to neighbors that no more messages are forthcoming.

- Ack is a message sent by a node in response to a Shutdown message. Its function is to signal to the node shutting down that the neighbor has received its previously sent Shutdown message, and therefore all messages sent previously.

5.4.3 Additional Node State

From the perspective of a particular node, say u, a neighboring node u' can be in one of three states: *unknown*, *blocked*, or *unblocked*. If the neighbor is blocked, u' is in the set `blocked` maintained by u; if it is unblocked, u' is in the set `unblocked`. A neighbor whose identifier is in neither of these sets is unknown. With an unknown neighbor, the only possible message that can be exchanged is Notify. Objects and object-related messages cannot be sent to blocked neighbors, but can be received from them.

When a node enters TRANSACT, it uses a set of node identifiers `sent_prepare` to keep track of the neighbors which have been sent a Prepare message. To keep track of which of these neighbors have replied in the form of Ready messages, it uses a set `recd_ready`. When a node enters ABORT, it starts to send out Abort messages to unblocked neighbors to which it has previously sent Prepare. To keep track of where it has sent Abort messages, the node uses the set `sent_abort`. Finally, when a node enters SHUTDOWN, it uses the sets `sent_shutdown` and `recd_ack` for bookkeeping on which neighbors have been sent Shutdown messages and from whom Ack has been received as a response.

5.4.4 Dynamics

In its early phases, the protocol largely follows the two-phase commit structure. Figure 5.4 shows fragments of an idealized, round-oriented execution of the protocol. Due to the inherent asynchrony in the network model, the protocol cannot be assumed to always execute in this way, but it highlights some key ideas.

Figure 5.4(a) shows a network of four nodes, where one node, u_0, has initiated the shutdown process by entering the state TRANSACT and sending Prepare messages to all its (known) neighbors u_1, u_2, and u_3. In Figure 5.4(b), all the neighbors have replied with Ready messages, since they were all in state IDLE. When u_0 has received all these messages, it enters state CLEAR and sends away all objects and object-addressed messages, as illustrated in Figure 5.4(c). When all objects have

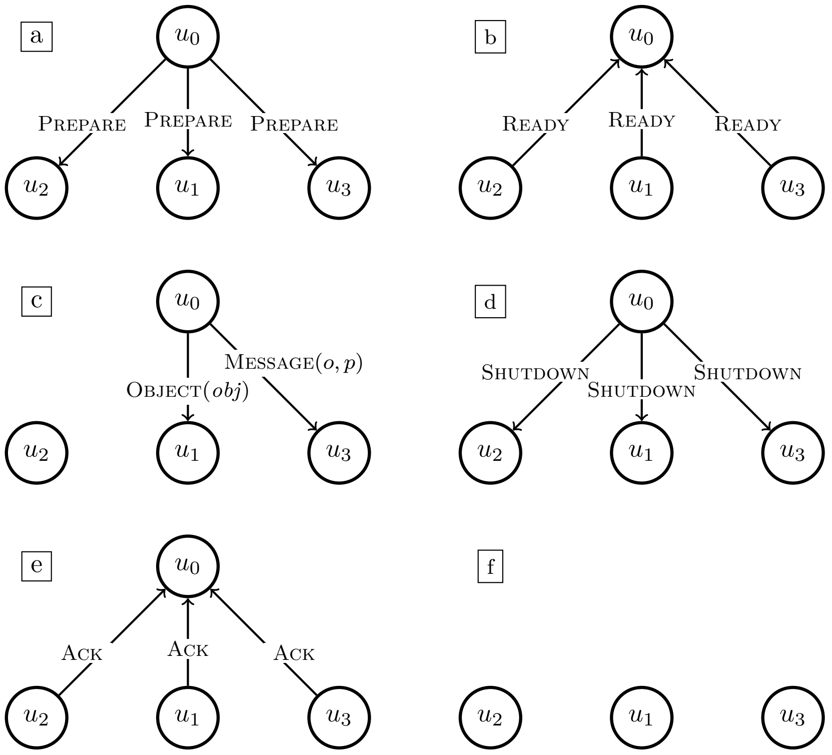


Figure 5.4: Idealized rounds of the shutdown protocol

been migrated and all object-related messages sent away, u_0 enters SHUTDOWN and sends Shutdown messages to all its neighbors, as in Figure 5.4(d). Neighbors confirm receiving this message by sending back an Ack message, shown in Figure 5.4(e). Finally, after u_0 has received all such messages, it shuts down, with the resulting network shown in Figure 5.4(f).

The main complexity of the protocol stems from dealing with possible interleavings of nodes in the process of shutting down. When a Prepare message arrives, the receiver can itself be in state TRANSACT, even if unblocked by the sender at the time the message was sent. Unless this tie between processes who are shutting down is resolved, with one node postponing shutdown until the other finishes, nodes may be unable to send their objects away—the only available neighbor may be the competitor. As a tie breaker, we rely on a relation, $<$, between node identifiers. At any time a node u is added, we assume that, for all existing nodes u' in the network, we have $u' < u$. Then, when there is a tie between two nodes, the node which is "less than" the other, according to the relation, wins. In this way, no node can be permanently unable to shut down by repeatedly losing tie breakers. In an implementation, it may be sufficient to have probabilistic guarantees that new nodes will lose tie breakers against existing nodes to get satisfactory results.

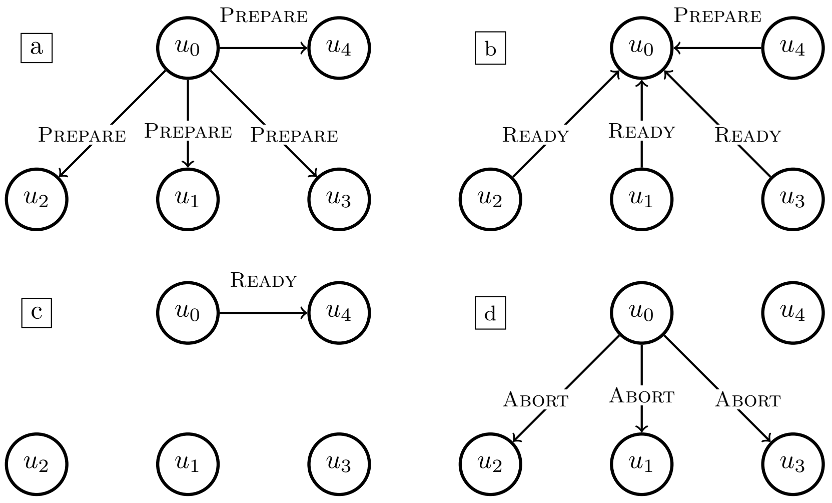


Figure 5.5: Idealized rounds of the shutdown protocol, with tie breaking

Figure 5.5 illustrates how tie breaking works. In Figure 5.5(a), the node u_0 has started the process of shutting down and sent PREPARE to all its known neighbors, which are unblocked as far as can be told locally. In Figure 5.5(b), it has turned out that the node u_4 is also in state TRANSACT, and has won the tie breaker with u_0, i.e., $u_4 < u_0$. Consequently, in Figure 5.5(c), u_4 has sent a PREPARE message of its own to u_0, while all other nodes are idle and have sent READY. After u_0 receives the PREPARE message, it enters state ABORT and sends ABORT messages to all who previously replied READY, as shown in Figure 5.5(d).

5.4.5 Definition of Node Behavior

We define how a node running the shutdown protocol behaves semi-formally by specifying how it mutates its state and transmits messages based on state predicates and incoming messages. This gives a definition that is close to implementable code, but which depends on assumptions about the network environment that are not fully enunciated. Hence, this view of the protocol is complementary to the formal models presented in Section 5.5. Besides the variables of sets of identifiers of neighbors mentioned above, nodes maintain a routing table `table`, a state indicator `state`, and use a special variable **`self`** to access their own (globally unique) identifiers.

The general syntax, in BNF format, of a predicated node action is shown in Listing 5.1. A *statement* is either an assignment of a value to a variable, the sending of a message to a neighbor, the **`if`** control construct, or the primitive operation **`shutdown`**. Except for **`if`** blocks, statements are ended by semicolons.

```
upon <predicate> [/\ <predicate>]*
do { <statement>* }
```

Listing 5.1: Format of node actions

Listing 5.2 shows a fragment of the protocol definition; the complete definition is listed in Appendix C. The functions `in()` and `union()`, correspond to the standard set operations of membership and union. We write `singleton(u)` for the singleton set containing only the element `u`. The symbol `~` is used for predicate negation. Note that all statements inside a **do** block are assumed to be executed atomically.

```
upon state = IDLE /\ receive Notify from u
  /\ ~ in(u, union(unblocked, blocked))
do {
  unblocked := union(unblocked, singleton(u));
  send Notify to u;
}

upon state = TRANSACT /\ receive Prepare from u
 /\ ~ u < self
do {
  if ~ in(u, sent_prepare) {
    sent_prepare := union(sent_prepare, singleton(u));
    send Prepare to u;
  }
}
```

Listing 5.2: Fragment of the shutdown protocol definition

5.5 Shutdown Protocol Analysis

The general idea of the correctness of the protocol in terms of safety, is that objects and object-related messages are always preserved on active nodes when a node shuts down. Intuitively, at each distributed network execution step, the *projection* of object-related state, including messages, is valid according to the underlying semantics of object behavior as described by the relation $\xrightarrow{ob}$. Hence, assuming steps that do not affect objects or object-related messages are disregarded from, the projected evolving object state corresponds to an abstract, network-oblivious program execution.

For correctness in terms of liveness, there are two separate notions of progress, namely, progress in execution of object tasks, and progress in nodes shutting down. The former depends on maintaining a connected network graph so that object-addressed messages can be delivered, and is discussed in Section 5.6, while the latter depends on the absence of hindrances to the shutdown process, which we concentrate on here.

In summary, at a reasonably rigorous level, correctness of the protocol may be specified as follows:

Safety For all nodes u in all valid network configurations, if u is in state SHUTDOWN and all nodes known to u are in `recd_ack`, then, u has no local objects, and for all nodes u' adjacent to u, there are no OBJECT or MESSAGE messages outstanding from u to u' or from u' to u.

Liveness In all valid configurations in strongly fair executions, for all nodes u, if u is in state TRANSACT, then u will eventually shut down. In addition, in all such configurations, for all nodes u, if u' is blocked at u, then u' eventually either becomes unblocked at u, or becomes unknown, i.e., neither blocked nor unblocked at u.

5.5.1 Promela Model

We model the protocol in the language Promela of concurrent processes, to enable verification in the on-the-fly model checker Spin [95]. The model is restricted to capturing the behavior of the protocol in a three-node fully connected network, to enable effective state space exploration.

Each of the three network nodes is a Promela process, which communicates with its neighbors through separate buffered channels. Three constants are defined to represent the three states in which a neighbor can be from the view of a specific node: `ADJACENT_UNKNOWN`, `ADJACENT_BLOCKED`, and `ADJACENT_UNBLOCKED`. A node maintains a **`byte`** array to keep track of these neighbor states. In addition, a node uses **`bool`** arrays in place of (mutable) sets of node identifiers. The abstract node states, e.g., IDLE, are represented as labels in the code. We abstract from objects and routing tables to minimize the state space, and thus only use the shutdown-related messages, i.e., we define

```
mtype = {notify,prepare,ready,abort,shutdown,ack};
```

Correctness in the model is specified by way of **`assert`** statements in the code. Verification then ensures that in all possible executions, these assertions hold. For example, if a process is in state TRANSACT, and receives a PREPARE message from its first neighbor with identifier `fst`, we assert that this neighbor is unblocked, as in Listing 5.3. This ensures that PREPARE messages can never arrive without the receiver being aware of the sender. This code fragment corresponds to the last node action definition in Listing 5.2.

One deviation from the protocol definition in Appendix C is in the use of process identifiers for tie breaking. In the definition, a newly added node must assume an identifier that is greater than all existing identifiers, to prevent scenarios where some node is never able to shut down. In the Promela model, when a node has shut down, it returns with the same identifier as before, albeit with other state reset. This

```
TRANSACT:
if
/* ... */
  :: recv[id].vector[fst]?prepare;
     assert(adjacent[fst] == ADJACENT_UNBLOCKED);
     if
       :: fst < id;
          /* ... */
       :: else;
          if
            :: sent_prepare[fst] == false;
               recv[fst].vector[id]!prepare;
               sent_prepare[fst] = true;
               goto TRANSACT;

            :: else;
               goto TRANSACT;
          fi;
     fi;
/* ... */
fi;
```

Listing 5.3: Fragment of the Promela model of the shutdown protocol

limits the applicability of the Promela model to verification of safety properties. Combined, the described abstractions in the model mean that verification in Spin only provides partial evidence of correctness.

The Promela model of the protocol has been checked with Spin version 6.3.2, and is available electronically [157].

5.5.2 Transition System Model

Model checking of bounded state spaces is insufficient to completely verify protocols that involve networks with arbitrary upper limits on the number of nodes; in principle, the network model used here allows continuous addition of neighbors to existing nodes nondeterministically. Therefore, we model the key parts of the shutdown protocol as a transition system by specifying formally network runtime configurations and a reduction relation. A runtime configuration is a collection *net* of nodes and arcs, with nodes defined as tuples without routing tables or objects, as follows:

$$\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{SHUTDOWN}, U_{rdy}, U_{prep}, U_{abrt}, U_{shtdn}, U_{ack})$$

Sets of identifiers U correspond to the values of the variables for tracking neighbors and messaging, in the obvious way. For brevity, we from sometimes write $\overline{U}_n$ for the node-related sets U_{unblk}, U_{blk}, and $\overline{U}_m$ for the message-related sets U_{rdy}, U_{prep}, U_{abrt}, U_{shtdn}, U_{ack}, or sublists of them, as determined by context. We also sometimes disregard from the exact order of sets inside node tuples.

A *valid* configuration is either the empty configuration ϵ or a configuration that can be reached by applying the reduction rules, modulo associativity and commutativity of configuration composition. The intention is to capture the protocol in an inductive framework, to enable accurate representation of higher-order datatypes such as finite maps and sets, and proofs of properties of valid configurations using induction.

The syntax and rules are specified in the language of the Ott tool [185], and then exported to the Gallina specification language of the Coq proof assistant [43]. Through the use of practical higher-order data structures, notably finite sets, the reduction rules implicitly define state transition functions for nodes, similar to the semi-formal definition in Appendix C. If defined inside the proof assistant, these functions can be extracted into, e.g., the OCaml programming language [119], and executed separately. However, this requires the use of possibly unverified external libraries for message passing.

Figure 5.6 shows the reduction rules that correspond to the node actions defined in Listing 5.2. Note that the **`if`** construct prompts the formulation of two separate rules.

(Recv-notify-unknown)

$$\frac{\texttt{first}\,(q) = \textsc{Notify} \quad \texttt{dequeue}\,(q) = q'' \quad u' \notin U_{unblk} \cup U_{blk}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\,\mathsf{ar}\,(u, q', u') \to \\ \mathsf{ar}\,(u', q'', u)\,\mathsf{nd}\,(u, U_{unblk} \cup \{u'\}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\,\mathsf{ar}\,(u, \mathsf{enqueue}\,(q', \textsc{Notify}), u')\end{array}}$$

(Transact-recv-prepare-nlt-in-prep)

$$\frac{\begin{array}{l}\texttt{first}\,(q) = \textsc{Prepare} \quad \neg\, u' < u \\ u' \in U_{prep} \quad \texttt{dequeue}\,(q) = q'\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{prep}, \overline{U}_m) \to \\ \mathsf{ar}\,(u', q', u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{prep}, \overline{U}_m)\end{array}}$$

(Transact-recv-prepare-nlt-notin-prep)

$$\frac{\texttt{first}\,(q) = \textsc{Prepare} \quad \texttt{dequeue}\,(q) = q'' \quad \neg\, u' < u \quad u' \notin U_{prep}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{prep}, \overline{U}_m)\,\mathsf{ar}\,(u, q', u') \to \\ \mathsf{ar}\,(u', q'', u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{prep} \cup \{u'\}, \overline{U}_m)\,\mathsf{ar}\,(u, \mathsf{enqueue}\,(q', \textsc{Prepare}), u')\end{array}}$$

Figure 5.6: Fragment of the shutdown protocol transition system definition

Consider a node u' that is in state SHUTDOWN. In the Promela model, once a process u receives an Shutdown message from u' and replies by sending Ack, blocking of u' at u ceases, and u' becomes unknown there. This means that, subsequently, u may send a Notify message to u', and thus block u' again, before u' has received the Ack message and shuts down. In a real execution, this normally means that u' is blocked forever for u, and, consequently, that u is itself never able to shut down. In the Promela model, this problem does not become apparent, since a node that shuts down can reappear after re-initializing its state, and then receive the Notify message. Note that the problem is not solved by introducing yet another message from u' to u, since there is no way for u' to know when u has received this message and finally shut down, except by receiving another reply from u.

In the transition system model, we solve this problem by introducing a transition that simultaneously removes a node and purges its identifier from all neighbors, thus exhibiting behavior similar to a reliable broadcast [26]. In effect, this means we introduce some synchrony into the model that must be accounted for at lower layers in an implementation.

In an accompanying Coq metatheory to the transition system definition, we establish a number of properties of the transition system that are necessary for protocol correctness by using induction on the type of the reduction rules. Notably, we establish that, in valid configurations, two distinct nodes which are unknown to one another never have outstanding messages in either direction, and characterize precisely the possible outstanding messages when nodes are in the process of exchanging Notify messages. However, this development falls short of full verification of safety, which we leave to future work. In addition, formulating and verifying liveness properties in Coq, by considering inductive predicates on infinite sequences via coinductive types [58], is a desirable extension of the metatheory.

Appendix D lists the complete runtime syntax and reduction rules. The complete definition in Ott, and the Coq proof scripts for the metatheory, which requires Coq version 8.4 and ssreflect version 1.5RC1 [137], is available electronically [157].

5.6 Maintaining a Connected Network Graph

The semi-formal protocol definition and the formal transition system model are maximally deterministic; they do not prescribe when, and in which situations, a certain node should shut down. Hence, the protocol allows executions where the network becomes partitioned into separate components. As a result, object tasks may become unable to progress, if they require messages from some object no longer located in the same network component. To always ensure the possibility of progress, node shutdown nondeterminism must be restricted to maintain network connectedness.

In general, whether the removal of a particular node will result in a partitioning of the network cannot be decided using only local neighborhood information. Consider the extreme case of a ring network; without prior knowledge, the only way a specific node in the network can get to know that both its neighbors are, in fact, still connected after the node has disappeared, is through a message passing chain that involves every node in the network.

5.6.1 A Simple Sufficient Condition of Connectivity

In a network with relatively high connectivity, e.g., hypergraphs, neighborhood connectivity information can be sufficient to ensure connectedness. More specifically, if a node in the process of shutting down knows the links of all of its neighbors, a sufficient condition of connectedness of the network without the node is that the neighbors by themselves form a connected subgraph. Figure 5.7 contrasts graphs with and without this condition for the node u_0. In the graph fragment in Figure 5.7(a), the subgraph of neighbors of u_0 only has only one (undirected) edge, between u_1 and u_3, leaving those nodes disconnected from u_2. In Figure 5.7(b), the subgraph of neighbors of u_0 is a linear, and thus connected, graph consisting of the nodes u_1, u_2, u_3, and u_4; its edges are emphasized by their thickness, as in the previous fragment.

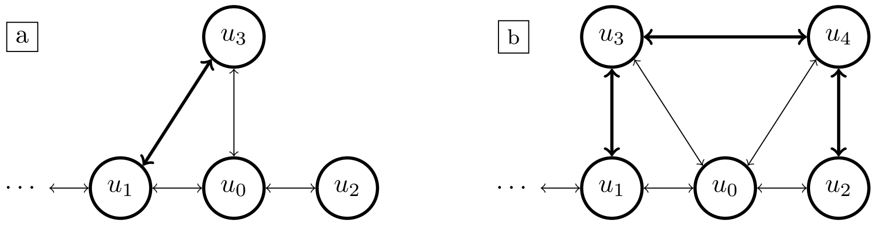


Figure 5.7: Graph fragments with and without high local connectivity

This suggests a modification to the protocol that is sufficient to ensure connectedness, but, on the other hand, sometimes unnecessarily constrains the shutdown process. Suppose Ready messages are changed to include a payload with the identifiers of all known (active) neighbors of the sender. Then, once the node attempting to shut down has received all such Ready messages, and it determines that the neighbor subgraph is connected, it can proceed by entering the CLEAR state. Otherwise, the node enters ABORT.

Assume the node u is shutting down, and has received Ready from its neighbor u'. Then, by the way the protocol works, u' cannot itself start to shut down until either u has finished the shutdown process, or u' has received Abort from u. Hence, when all Ready messages have been received by u, the subgraph of its neighbors will not change unless u takes some action. Accordingly, if the subgraph is connected, it will remain so at least until u has shut down.

5.6.2 General Local Tests of Connectivity

A graph is k-connected when any set of $k-1$ nodes can be removed without disconnecting the graph. The above proposed check for the simple sufficient condition of connectivity is in fact a special case of a natural local test of graph k-connectivity described by Cornejo and Lynch [45]. The idea is to construct, through round-based communication, the subgraph of all nodes at a certain distance from the node under consideration. If this subgraph is k-connected, then the whole graph is also k-connected.

The problem with this test in the present setting is the reliance on synchronous rounds, and the fact that only immediate neighbors are prevented from shutting down themselves if a node successfully reaches state CLEAR. The former problem can be solved by instead relying on asynchronous echo, or wave-propagation, algorithms

[123], limited to specific depths. The latter problem calls for extending the protocol with transactions spanning all nodes at distances longer than one hop. We leave detailed descriptions of these extensions as future work.

5.7 Strategies for Decentralized Power Control

The shutdown protocol definition, as given in Section 5.4 and Section 5.5, is maximally nondeterministic, both with respect to object mobility and node shutdown decisions. This nondeterminism leaves room for using decentralized strategies to perform adaptation in both how objects distributed between nodes, but also in whether nodes are turned on and off in response to requirements on throughput and energy consumption.

5.7.1 Adaptability Objectives

As in Chapter 4, we define the load of a particular node as the number of objects that are executing on it. We want to achieve *balanced* allocations of objects to nodes, i.e., allocations where the objects are distributed as evenly as possible among all nodes.

Define l_{avg} as the average load of nodes in a network configuration determined by context. Suppose two constants l_{min} and l_{max} are given, such that $0 < l_{min} \leq l_{max}$. A possible objective of node and object adaptability is then to achieve (1) balanced allocations of objects to nodes, where (2) $l_{min} \leq l_{avg} \leq l_{max}$. Note that this precludes networks with "too many" nodes, i.e., networks where a balanced allocations would result in fewer objects than l_{min} per node, and also networks with "too few" nodes, where balanced allocations would result in more objects than l_{max} per node. The values of these constants would be determined based on desired overheads and their costs in terms of, e.g., energy.

5.7.2 A Node Coin-Flip Heuristic for Power Control

Building on the approach in Chapter 4, we can define a heuristic for decentralized addition and shutdown of nodes. At startup, a node sets up a distributed aggregation process [54] to continually receive the global load average, l_{avg}. Due to churn among objects and asynchrony, this will at best be a reasonable estimate of the quantity.

Consider first the case of networks having too many nodes. Suppose we have a node u in state IDLE with load l_u, $l_u < l_{min}$, which has become aware that $l_{avg} < l_{min}$. Then, this node flips a coin and starts to shut down with probability $1 - l_u/l_{min}$. Consequently, if u has much lower load than l_{min}, the probability of shutdown is close to 1, while a load close to l_{min} gives a low probability. To avoid an overwhelming number of shutdowns due to an uneven distribution of load, e.g., due to objects being created at a single node faster than they are migrated away, nodes can set up an aggregation process for the load variance or standard deviation; when this measure is above a certain threshold, no shutdown coin-flips are made.

Assume an active node u can, by using some atomic operation, start up an inactive neighboring node u'. Suppose again u is in IDLE, but with load l_u such that $l_u > l_{max}$. Whenever u finds out that $l_{max} < l_{avg}$, it flips a coin and starts up u' with probability $1 - l_{max}/l_u$. As above, to avoid starting up inordinately many nodes in a short interval due to uneven load distribution, a measure of variance can be used as a threshold.

5.8 Related Work

Two-phase commit protocols were first discussed in the context of database systems [82, 117, 187]. The shutdown protocol differs in several important ways from these protocols, not least in its network model. In two-phase commit protocols, there is a specific *leader* that initiates a distributed transaction; in the shutdown protocol, all nodes are potential initiators of a shutdown process. The leader in a two-phase commit expects a yes/no answer to a transaction request, while in the shutdown protocol, only “yes” answers are explicit. Negative, “no” answers are given in the form of a competing transaction initiation request. Instead of resilience against site failures, i.e., crashes, the shutdown protocol enables resilience against simultaneous shutdown of neighboring nodes, and the subsequent potential loss of objects and object-related messages.

Field and Varela [69] propose a programming model based on actors, called Transactors, that is tolerant to crash failures through *checkpointing* of state. The model assumes failing nodes either stop completely or revert to a previous checkpoint, saved to reliable storage. In contrast to the present work, which is more implementation-oriented, node behavior and state is only considered indirectly, through the semantics of transactors. As in the Actor model generally [3], the message passing medium is also implicit. Crash tolerance and controlled shutdown are largely complementary; the latter can lead to lower overheads than, e.g., recovery, and is thus to be preferred, when possible.

De Nicola et al. [152] extend the Klaim language [19] to account for locations and network topologies. The resulting language, TKlaim, is enriched with failures of both nodes and links. By allowing modifiable connections, TKlaim can capture process execution in dynamic networks. A distinct difference from our network model is the use of a tuple spaces for messages and shared data.

Cornejo et al. [44] describe a distributed algorithm for ensuring connectivity of mobile agents while they perform some task. The algorithm relies on 1-hop broadcast among agents for communication, and therefore does not require a routing infrastructure. In contrast to the present network model, the algorithm assumes that the physical distance of agents affects communication, which is performed in synchronous rounds. Cornejo and Lynch [46] prove that there exists no local graph trait which perfectly captures graph k-connectivity.

5.9 Conclusion

We have presented a decentralized protocol for orderly shutdown, and addition, of nodes connected point-to-point using asynchronous message passing channels, which host communicating, mobile distributed objects whose integrity must be preserved. The protocol is part of ongoing work on implementation correctness and efficient resource allocation for active objects in networks, as described in Chapter 2, Chapter 3, and Chapter 4.

As future work, we plan to complete the verification of protocol correctness in the transition system model which allows networks to grow and shrink arbitrarily, both in the form of safety and liveness. We also plan to implement the protocol inside our existing simulator, described in Chapter 4, for networked execution of ABS programs, and evaluate and refine the proposed strategies for decentralized power control in conjunction with load balancing of objects. Additionally, the proposed protocol extensions that ensure network connectivity must also be proven formally correct and evaluated in practice.

We aim to extend our correctness analysis to dynamic networks with crash failures, using replication [69] and failure detectors [36, 150]. By adopting techniques similar to those used for security of "passive" objects in content-centric networking [188], along with inlined monitors and proof-carrying code [132], the model can become resistant to Byzantine failures. Finally, other implementation-level concerns, such as buffer management, garbage collection, and compactness of location independent routing tables [186], remain to be addressed.

Acknowledgements

We thank Andreas Lundblad for constructive discussions and criticism.

Chapter 6

Dynamic Probabilistic Inference of Atomic Sets

Peter Dinges[1] Gul Agha[1] Karl Palmskog[2]

[1]University of Illinois at Urbana-Champaign, USA
pdinges@acm.org, agha@illinois.edu
[2]KTH Royal Institute of Technology, Sweden
palmskog@kth.se

Abstract

Concurrent programs often ensure the consistency of their data structures through synchronization. Because control-centric synchronization primitives, such as locks, are disconnected from the consistency invariants of the data structures, a compiler cannot check and and enforce these invariants—making it hard to detect bugs caused by incorrect synchronization. Moreover, a consistency bug may be the result of some unlikely schedule and therefore not show up in program testing. In contrast, data-centric synchronization adds annotations to data structures, defining sets of fields that must be accessed atomically. A compiler can check such annotations for consistency, detect deadlock, and automatically add primitives to prevent data races. However, annotating existing code is time consuming and error prone because it requires understanding the concurrency semantics implemented in the code. We propose a novel algorithm, called Bait, for deriving such annotations automatically from observed program executions using Bayesian probabilistic inference. The algorithm produces atomic set, unit of work, and alias annotations for atomic-set based synchronization. Using our implementation of the algorithm, we have derived annotations for large code bases, for example the Java collections framework, in a matter of seconds. A comparison of the inferred annotations against manually converted programs, and two case studies on large, widely-used programs, show that our implementation derives detailed annotations of high quality.

6.1 Introduction

A program that synchronizes every single field access for each concurrently used object can still exhibit *high-level* data races [12, 25]: fields are often connected through invariants and must be updated together to maintain the object's consistency [200, 127]. For example, the value of the `size` field of a list object must equal the number of elements in the array that stores the list entries. Interleaved access to such fields from concurrent threads can expose or produce an inconsistent state in the object containing those fields.

High-level data races may be prevented with control-centric synchronization mechanisms such as *locks*. However, to protect a group of data fields, the programmer must recognize all execution paths that result in problematic interleavings, and use locks to prune them. This requires complicated non-local reasoning over all possible execution paths.

An alternative is to use *data-centric* synchronization [200, 62], which localizes the reasoning by asking the programmer for annotations that specify which fields of an object are connected by a semantic invariant. A compiler can use these annotations to add primitives that prevent interleaved access to fields in the same semantic unit. This reduces the potential for high-level data races on execution paths that the programmer may not have conceived of. Furthermore, the annotations can be statically checked for consistency [62] and deadlock-freedom [134].

Experience with converting a set of concurrent Java programs to data-centric synchronization shows that such annotations are sufficiently expressive to represent the desired semantic properties, and that the approach may achieve good performance [62]. However, while the end-results are encouraging, the conversion process itself is time-consuming: it can take several hours even for a relatively small and simple program. The difficulty in doing such a conversion lies in the need to understand the program's concurrency semantics, which can be complicated even for small code sizes. For large legacy programs, understanding the concurrency semantics is likely a daunting challenge.

There are two kinds of problems that can result in problematic annotation: first, unrelated fields may be connected by annotations; second, connections may be omitted. The first type of error reduces the available concurrency, the second type can result in incorrect execution, for example due to high-level data races. On careful examination, we found both problems in the six programs that had been manually converted in prior work [62]. In two cases, the annotations accidentally introduce a global lock, and in two other cases, annotations for shared object synchronization are omitted.

In previous work [60], we proposed and implemented a dynamic inference algorithm for the automatic conversion of shared memory multi-threaded programs from control-centric to data-centric synchronization. The algorithm infers annotations for data-centric synchronization that is based on *atomic sets* [200, 62]. It uses simple set membership criteria for classification. While this algorithm produces results that are apparently comparable or better than manual annotation, the algorithm is brittle (affected by small perturbations in execution traces) and not scalable to long executions. Based on our experience with evaluating this algorithm, we concluded that an effective algorithm for dynamic annotation inference should:

- take into account the *distance* between two related observations in terms of basic operations to distinguish between unrelated computation phases;
- produce the same annotations for similar input traces, regardless of minor fluctuations like occurrences of rare data races;
- support long executions by bounding the observation data by a size determined by the code base, rather than the duration of the execution;
- improve the accuracy of estimates as more data becomes available.

The contribution of this paper is a novel probabilistic reasoning algorithm using *Bayesian inference* which exhibits the above properties. The algorithm, called Bayesian Annotation Inference Technique (Bait), can continually adjust to new evidence from traces, and handle large programs and long executions. We describe the algorithm and demonstrate its effectiveness firstly by comparing inferred annotations to manual annotations added in prior work for six programs, and secondly in two case studies on large, widely-used programs. The inferred concurrency semantics enables safe program execution because it automatically prevents schedule-dependent *Heisenbugs*. Such bugs are (by definition) unlikely to manifest during testing.

6.2 Background: Atomic Sets

Our implementation of Bait infers annotations for data-centric synchronization based on *atomic sets* [200, 62]. An atomic set is a group of data fields inside an object that are connected by a *consistency invariant*. Objects can contain multiple disjoint atomic sets. Recall the list example in Section 6.1: the value of the list's `size` field must equal the number of elements in the `elements` array used to store the list entries. Thus, the fields `size` and `elements` form an atomic set. Listing 6.1 shows the respective code and annotations in AJ [62], a Java dialect that supports atomic sets in addition to control-centric constructs like **`synchronized`** blocks. The **`atomicset`** statement in line 2 declares an atomic set `L`; the **`atomic`**`(L)` annotations of the field declarations add the fields `size` and `elements` to `L`.

Instead of requiring an explicit expression of a consistency invariant, for example `size == elements.length`, an atomic set is complemented by one or more *units of work*. A unit of work is a method that preserves the consistency of its associated atomic sets when executed sequentially. Thus, atomic sets can ensure the application's consistency by inserting synchronization operations that guarantee the sequential execution of all units of work. By default, all non-private methods

```
 1 class List {
 2   atomicset L;
 3   atomic(L) int size;
 4   atomic(L) Object[] elements;
 5   public int size() {
 6     return size;
 7   }
 8   public Object get(int index) {
 9     if (0 <= index && index < this.size)
10       return this.elements[index];
11     else
12       return null;
13   }
14   public void addAll(unitfor(L) List other) {
15     this.size = this.size + other.size;
16     /*...*/
17   }
18   /*...*/
19 }
20
21 class DownloadManager {
22   atomicset U;
23   atomic(U) List urls|L=this.U|;
24   public URL getNextURL() {
25     if (this.urls.size() == 0) return null;
26     URL url = (URL) this.urls.get(0);
27     this.urls.remove(0);
28     announceStartInGUI(url);
29     return url;
30   }
31   /*...*/
32 }
```

Listing 6.1: Sample classes in the AJ dialect of Java, which adds data-centric synchronization via the annotations **`atomicset`**, **`atomic`**, **`unitfor`**, and `|A=`**`this`**`.B|`.

of a class are units of work for all atomic sets declared in the class or any of its subclasses. Like field declarations, atomic sets use classes as scopes, but are instance specific at runtime.

Consider the method `get(`**`int`**`)` in Listing 6.1. The method is a unit of work for the atomic set `L` of its containing `List` object. The atomic sets of two `List` objects are distinct. Other methods can be declared units of work with the **`unitfor`** annotation. In line 14 of Listing 6.1, the method `addAll(List)` is not only a unit of work for the atomic set `L` of its own `List` object, but also for the atomic set `L` of its argument. Hence, two threads, t_1 and t_2, that concurrently invoke `get(`**`int`**`)` and `addAll(List)` on a `List` l cannot interleave when accessing l's field: either t_1 executes `get(`**`int`**`)` first, or t_2 executes `addAll(List)` first. The interleaved case where t_2 has updated l`.size` but not l`.elements`, which causes t_1 to violate the array bounds cannot occur.

Aliases extend atomic sets beyond object boundaries. An alias merges the atomic set containing a field with an atomic set in the object that is the field's value. For example, consider the `DownloadManager` class in Listing 6.1. The alias annotation `|L=`**`this`**`.U|` of the `urls` field declaration combines the atomic set `L` with the atomic set `U`. Hence, the `getNextURL()` method is a unit of work for this combined atomic set; its access to the `urls` list cannot be interleaved. This guarantees that no other thread can empty the list between the invocations of `get(`**`int`**`)` and `remove(`**`int`**`)`.

Coarse-grained atomic structures like trees can be implemented with aliases. AJ supports aliasing of elements in an array, and even aliasing of atomic sets in the elements of arrays. Additionally, AJ allows for more advanced annotations such as partial **`unitfor`** declarations, **`fastread`**, and **`internal`**. These simplify the specification of units of work and help an AJ compiler generate more efficient concurrency-control code. The more advanced annotations are secondary; we do not consider their inference in this paper.

6.3 Algorithm Ideas by Example

We now explain the key concepts in Bait using the example program shown in Listing 6.2. The program downloads files in parallel, managing its network connections via threads. It is a version of the program in Listing 6.1 that uses control-centric synchronization; our goal is to infer the AJ annotations shown there.

Bait monitors the execution of the program. It observes event sequences, such as the one partially displayed in Figure 6.1, and infers data-centric synchronization annotations based on the following intuitions: (1) the fields of an object that a thread accesses together, without interleaving, should belong to the same atomic set; and, (2) groups of objects that a thread accesses together should be connected by aliases.

In the partial execution shown in Figure 6.1, one of the downloading threads invokes `getNextURL()` to request a new URL to download from the shared manager. After ensuring that the list of pending URLs contains an entry (line 23), the manager picks and removes the first one. The manager then announces the start of the download in the program's user interface (line 26) and finally returns the value to the thread.

6.3.1 Inference of Atomic Sets

Bait assumes that the methods of a program perform semantically meaningful operations and that the trace during an execution (mostly) represents the intended behavior of the program—for example, such a trace may be generated by running an existing test suite.

Given these assumptions, the fields of an object accessed atomically by a method in close succession are likely connected by some invariant. The set of fields that a method accesses atomically is consequently a *candidate* atomic set; the method

```
 1 class List {
 2   int size; // Connecting invariant: elements[i] is valid if i < size
 3   Object[] elements;
 4   public int size() {
 5     return size;
 6   }
 7   public Object get(int index) {
 8     if (0 <= index && index < this.size)
 9       return this.elements[index];
10     else
11       return null;
12   }
13   /*...*/
14 }
15
16 class DownloadManager {
17   List urls;
18   public boolean hasNextURL() {
19     return this.urls.size() > 0;
20   }
21   public URL getNextURL() {
22     // Synchronization in run() makes call sequence atomic
23     if (this.urls.size() == 0) return null;
24     URL url = (URL) this.urls.get(0);
25     this.urls.remove(0);
26     announceStartInGUI(url);
27     return url;
28   }
29   /*...*/
30 }
31
32 class DownloadThread extends Thread {
33   DownloadManager manager;
34   public void run() {
35     while (true) {
36       URL url;
37       synchronized(this.manager) {
38         if (!this.manager.hasNextURL()) break;
39         url = this.manager.getNextURL();
40       }
41       download(url); // Blocks while waiting for data
42     }
43   }
44   /*...*/
45 }
```

Listing 6.2: Example downloading program that uses threads to manage multiple network connections. Threads share a single manager that maintains the list of URLs to download. The program uses control-centric synchronization (line 37); the goal is to infer the AJ annotations shown in Listing 6.1.

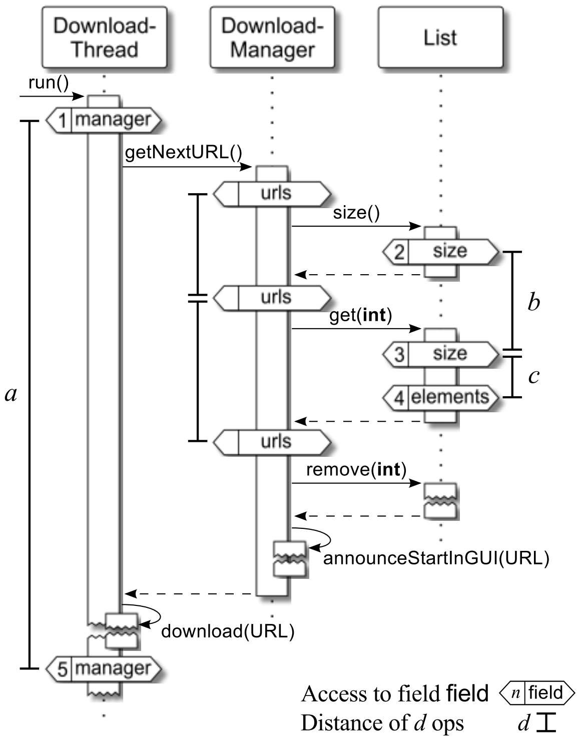


Figure 6.1: Sample execution of the program in Listing 6.2 (starting from line 39) that demonstrates the basic ideas of the algorithm.
Accessing fields close together and without interleaving from another thread, exemplified by accesses 3 and 4, suggests that an invariant connects the fields. Hence the fields `size` and `elements` likely belong to an atomic set; the method `get(`**`int`**`)`, in which the accesses occur, is a unit of work for this atomic set. In contrast, accesses far apart (1 and 5) or interleaved accesses (not shown) discourage an atomic set.
Close field accesses can occur in different methods, such as accesses 2 and 3. In this case, the event context is the lowest common ancestor of the methods in the call tree (here: `getNextURL()`). Observing close accesses to fields *within* a field, for example the two accesses to `urls.size` in `getNextURL()`, is evidence that there should be an alias from the atomic set that contains `urls` to the one that contains `size`.

itself is a candidate unit of work for this atomic set. For example, the `get(`**`int`**`)` method in Listing 6.2 reads the fields `size` and `elements` in the same list object. In the sample execution of Figure 6.1, the reads (accesses 3 and 4) happen close together and without interleaving. Thus, we have evidence that the class `List` should contain an atomic set with these two fields. Method `get(`**`int`**`)`, the *context* of the field accesses, is a unit of work for this potential atomic set.

However, field accesses within a method may be far apart. For example, the two accesses to the thread object's `manager` field in the `run()` method of `DownloadThread` (1 and 5) are separated by a method call with many operations. Observing a large *distance* like a between two field accesses diminishes the likeliness of an invariant between the fields. Such an observation hence counts as evidence *against* an atomic set containing the fields. The same is true for interleaved access to fields by multiple threads.

The central idea of the algorithm is to use this evidence for and against atomic sets in Bayesian inference. Collecting evidence, BAIT updates its *belief* that fields should be included in the same atomic set. If the belief is high enough at the end of the execution—intuitively, there was stronger evidence for an atomic set than against it—BAIT outputs corresponding **`atomic`** annotations.

6.3.2 Inference of Aliases

Since high-level semantic operations often employ low-level operations, field accesses may belong to different contexts. In Figure 6.1, access 2 happens within the `size()` method; access 3 happens within the `get(int)` method of `List`. Increasing the distance between the accesses ($b > c$) suffices to adjust the atomic set evidence in this case. However, the context that contains both accesses is no longer obvious.

The algorithm uses the lowest common ancestor in the call tree as context for field accesses belonging to different methods. For accesses 2 and 3, this is the `getNextURL()` method. Intuitively, we observe a pair of close atomic accesses to `urls.size` within that context. Besides being evidence for an atomic set containing field `size`, this suggests that `getNextURL()` is a unit of work for this atomic set. Because the method accesses `size` via the field `urls`, there should be an alias from the atomic set containing `urls` to the one containing `size`.

However, aliases can remove all concurrency from the program when they include objects shared between threads. In Figure 6.2, two download threads share a manager. Each thread's `run()` method is context for two close atomic accesses to the field `urls` in the manager object (accesses $6, 7$ and $8, 9$). Performing inference as above, this suggests an alias that merges the atomic set in class `DownloadThread` containing the `manager` field with the one in `DownloadManager` containing the `urls` field. The alias makes the `run()` method a unit of work for the manager's atomic set that contains the `urls` field. As a consequence, the execution of the `run()` methods must be sequentialized, which means that only one of the two threads can be active at all, reducing performance.

BAIT mitigates the sequentialization problem by tracking which objects threads access together and weakening the belief in aliases across the boundaries of such object clusters. In our example, both threads access themselves, the manager, and the list object. Thus, the heuristic detects three clusters of objects: two that are accessed by a single thread (the thread objects themselves), and one that is accessed by both threads (the manager and the list object). Maintaining the boundaries

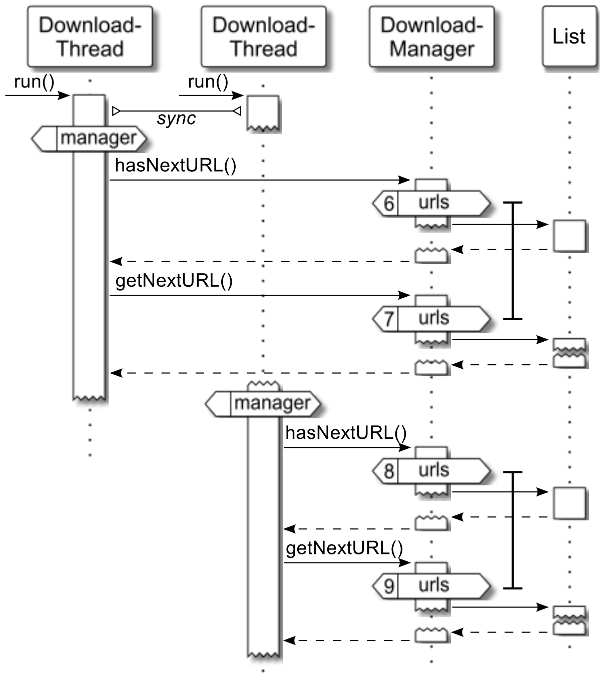


Figure 6.2: Sample execution of the program in Listing 6.2 that highlights a challenge in alias inference.
Both threads observe close atomic accesses $(6, 7$ and $8, 9)$ to `manager.urls`, which suggests an alias from the atomic set in class `DownloadThread` containing the `manager` field to the one in `DownloadManager` containing `urls`, compare Figure 6.1. Unfortunately, such an alias makes both `run()` methods into units of work that operate on the shared manager object, which prevents them from executing concurrently.
Bait mitigates the problem by tracking which objects threads access together and weakening the belief in aliases across the boundaries of such object clusters. In the sample execution, the heuristic detects three clusters of objects: two consisting of the individual thread objects, and one consisting of the manager and list object. It therefore discourages the problematic alias from `DownloadThread` to `DownloadManager`, but does not interfere with the alias from `DownloadManager` to `List`.

between these clusters, the heuristic prevents aliases from `manager` to `manager.urls` to preserve concurrency, but allows an alias from `urls` to `urls.size`.

6.4 Algorithm

This section describes Bait in detail, building upon the ideas explained in Section 6.3.

6.4.1 Field Access Observations

During the execution of a workload, the algorithm records observations of *get* and *put* operations on the fields of each object. These observations are captured in the scope of a method call for a thread. From two consecutive observations for the *same* object, Bait generates a *field access event* e, which is a tuple

$$(f, g, d, a) \in \mathrm{Fd} \times \mathrm{Fd} \times \mathbb{N} \times \mathrm{At}.$$

Here, Fd denotes the set of all fields in the program; f is the first field accessed, g is the second field. The distance d between the two accesses is the number of basic operations executed by the thread, such as Java byte code instructions. The entry $a \in \mathrm{At} = \{\mathrm{atomic}, \mathrm{interleaved}\}$ signals whether access to both f and g was atomic or access to g was interleaved with some other thread. To detect such interleaved accesses, Bait relies on a separate race detection algorithm such as FastTrack [74], which is used in the implementation described in Section 6.5.

6.4.2 Bayesian Detection of Semantic Invariants

Using the generated field access events, the algorithm aims to determine whether there are invariants that hold between pairs of fields. Consider two fields f and g accessed in method m of a thread when executing a program on some workload. Suppose the workload generates the events $e_1, \ldots, e_n$, all related to f and g. Write H for the hypothesis that there exists a semantic invariant connecting f and g in the method, and $\neg H$ for the negated hypothesis that there is no such invariant.

Our goal is to find out to what degree the *evidence*, in the form of $e_1, \ldots, e_n$, supports the conclusion that H holds. In the Bayesian probabilistic reasoning framework [163], this degree of support is formalized as the conditional probability of H given $e_1, \ldots, e_n$, which through Bayes's formula can be written as

$$P(H|e_1, \ldots, e_n) = \frac{P(e_1, \ldots, e_n|H) \cdot P(H)}{P(e_1, \ldots, e_n)}. \tag{6.1}$$

Unfortunately, the right-hand side is difficult to estimate because it would require guessing the absolute probability that the events $e_1, \ldots, e_n$ occur in a program. For estimation, it is more convenient to use relative values such as the so-called odds and likelihood ratios. Intuitively, the likelihood ratio expresses how many times more likely an event is when the hypothesis is true versus when the hypothesis is false. Thus, we divide the left-hand side of Equation 6.1 with its complementary form, yielding

$$\frac{P(H|e_1,\ldots,e_n)}{P(\neg H|e_1,\ldots,e_n)} = \frac{P(e_1,\ldots,e_n|H)}{P(e_1,\ldots,e_n|\neg H)} \cdot \frac{P(H)}{P(\neg H)}.$$

What the equation says is that our revised belief in H, when presented with $e_1,\ldots,e_n$, is equal to the ratio of the chances of observing $e_1,\ldots,e_n$ under H and $\neg H$, times our initial belief in H. We call the revised belief *posterior odds*, the ratio of the chances of making observations the *likelihood ratio*, and our initial belief the *prior odds*. More compactly, then, we write the equation as

$$O(H|e_1,\ldots,e_n) = L(e_1,\ldots,e_n|H) \cdot O(H). \tag{6.2}$$

These quantities are easier to estimate than probabilities, yet must be recomputed from scratch every time new evidence is added. However, if $e_1,\ldots,e_n$ are *conditionally independent* given H, an assumption discussed in Section 6.4.3, we have

$$P(e_1,\ldots,e_n|H) = \prod_{k=1}^{n} P(e_k|H),$$

and similarly for $\neg H$, which together with Equation 6.2 gives

$$O(H|e_1,\ldots,e_n) = O(H) \cdot \prod_{k=1}^{n} L(e_k|H).$$

This equation suggests that recursive, on-the-fly computation of odds is possible, as becomes clear when adding one more piece of evidence e_{n+1}, yielding

$$O(H|e_1,\ldots,e_n,e_{n+1}) = L(e_{n+1}|H) \cdot O(H|e_1,\ldots,e_n).$$

We set $O(H) = 1$, that is, we assume that H and $\neg H$ are initially equally likely. We have thus reduced the problem of obtaining the degree of support for H to computing $L(e|H)$, given the data from e.

6.4.3 Conditional Independence of Events

Conditional independence means that knowledge of H, or $\neg H$, makes evidence up to that point irrelevant with respect to future evidence. Equivalently, under conditional independence, the hypothesis influences the evidence directly, without systematic interference from external factors. However, in a run of the algorithm, the evidence produced can clearly be skewed through systematic influence from the chosen workload and the scheduler.

One way to address this problem is to refine the (coarse-grained) hypothesis space that either H or $\neg H$ holds into multi-valued variables [163]. This leads to a considerably more complicated mapping of evidence to likelihoods ratios. Instead of taking this route, we argue that the influence of external factors can be minimized by running BAIT on workloads with sufficient code coverage for long enough to exhibit all critical interleavings, if necessary using tools to instrument scheduling and execution paths.

Although BAIT can falsely conclude that two fields are related by an invariant (and thus include them in an atomic set or add an alias) when they are not, the resulting behavior is still safe. However, performance may suffer because of such an error, due to increased overhead from synchronization and reduction of concurrency.

6.4.4 Estimation of Likelihood Ratios

Suppose the field access event e reports we have a distance d between atomic accesses of f and g. Intuitively, the likelihood ratio $L(e|H)$ we assign based on e should have the following properties:

1. As d decreases, $L(e|H)$ must increase, but only up to some point, after which it becomes a flat maximum value; even if atomic accesses of f and g happen in close proximity, it is not conclusive that H holds.

2. As d increases, $L(e|H)$ must decrease, but only to some minimum value greater than zero; one observation should not make it impossible to conclude that H holds.

BAIT therefore uses a *logistic function* $\ell(d)$, as shown in Figure 6.3, to map field access events to likelihood ratios. For example, accesses 2, 3, and 4 in Figure 6.1 occur in close succession. We interpret this as evidence that it is more likely than not that an invariant connects the fields `size` and `elements`. Hence, we assign the distances b and c, with $b > c$, likelihood ratios $\ell(c) > \ell(b) > 1$. In contrast, the large distance a diminishes our belief that an invariant connects the two accesses to the `manager` field. Thus, we set $1 > \ell(a) > 0$. We leave the exact parameters of the logistic curve—its steepness and minimum and maximum likelihood ratios—to be determined during an implementation of the algorithm.

However, distance is not the only criterion for estimating the likelihood ratio. Suppose that e reports interleaved access. We then disregard the distance and set $L(e|H)$ to a real number p (“penalty”) close to zero. This reflects that, intuitively, our belief in an invariant goes down significantly after witnessing interleaving, while not making it impossible to infer the invariant’s existence later on, through overwhelming atomic access. BAIT is thus robust against sporadic errors like very rare data races. We again leave the precise value of p to an implementation.

In summary, given a field access event $e = (f, g, d, a)$, we define the estimated likelihood ratio for e as

$$\ell(d,a) = \begin{cases} \ell(d) & \text{if } a = \text{atomic;} \\ p & \text{if } a = \text{interleaved.} \end{cases}$$

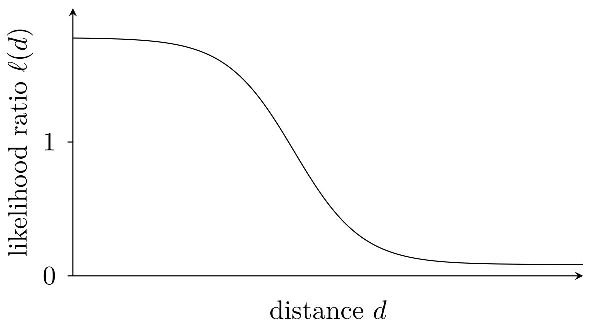


Figure 6.3: Logistic curve for mapping atomic-access distances to likelihood ratios

6.4.5 Belief Configurations

We can now define how BAIT stores odds of invariants and uses likelihood ratios to update these odds in the course of workload execution.

Odds are stored in *affinity matrices.* An affinity matrix A is a symmetric map from pairs of fields (f, g) to real numbers. Symmetric means that the value assigned to (f, g) equals the one assigned to (g, f). Setting x as the value of (f, g), written $A[(f, g) \mapsto x]$, maintains the symmetry: after the update, it is $A(g, f) = x$.

Belief configurations describe the algorithm's state. A belief configuration B contains an affinity matrix A_m for every method m. Recall that an access event for a thread t in method m is a tuple consisting of two fields $f, g \in \mathrm{Fd}$, a distance $d \in \mathbb{N}$, and an atomicity indicator $a \in \mathrm{At}$. The transition function for belief configurations

$$\delta_{t,m} : \mathrm{Config} \times \mathrm{Fd} \times \mathrm{Fd} \times \mathbb{N} \times \mathrm{At} \to \mathrm{Config}$$

is now defined as $\delta_{t,m}\big(B, (f, g, d, a)\big) = B[m \mapsto A'_m]$ with

$$A'_m = A_m\big[(f, g) \mapsto \ell(d, a) \cdot A_m(f, g)\big]. \tag{6.3}$$

For all methods m, define an initial affinity matrix A_m^{init} such that $A_m^{\mathrm{init}}(f, g) = 1$ for all $(f, g) \in \mathrm{Fd} \times \mathrm{Fd}$ and an initial belief configuration B^{init} with $B^{\mathrm{init}}(m) = A_m^{\mathrm{init}}$. Then, if the events $e_1, \ldots, e_n$ are generated in t for $m_1, \ldots, m_n$, the algorithm computes the final belief configuration

$$\delta_{t,m_n}(\cdots \delta_{t,m_1}(B^{\mathrm{init}}, e_1) \cdots, e_n),$$

which is ultimately used for deriving atomic sets for the program.

6.4.6 Inference of Aliases and Units of Work

Inference of aliases and units of work is done at the same time as inference of atomic sets, and in a similar way, but with several important differences.

Suppose we observe an atomic access of the field g after an access of f in the method m. Within m, the object that contains f and g may be known by a source code identifier, that is, by a field or parameter name n. For example, in Figure 6.1, the field access event generated for the accesses 2 and 3 occurs in method `getNextURL()`. Within that method, the list object that contains the accessed `size` field is known by the field name `url`. Hence the method observes the accesses in the context of this name, as `urls.size`; and more generally, m observes the accesses of f and g as $n.f$ and $n.g$.

Such an observation indicates that m performs multiple operations on another object (the list in our example). As before, if the distance d between the accesses $n.f$ and $n.g$ is small, then these operations likely maintain an invariant. Therefore, they should be atomic, which means that an atomic set containing n should be extended—by an alias—to also contain $n.f$ and $n.g$. Translated to our example, the close accesses to `urls.size` count as evidence for an alias that merges the manager object's atomic set containing `urls` with the list object's atomic set containing `size`.

In summary, to infer aliases and units of work, we associate with each identifier n an affinity matrix A_n, and update this matrix with the likelihood ratio $\ell(d)$, penalizing interleaved accesses as for atomic sets above. Then, most straightforwardly, if $A_n(f, g) > 1$ for A_n in the final configuration, this suggests an alias from the inferred atomic set of n—should n be a field name—to the inferred atomic set of f and g. Should n be a parameter of m, then this suggests declaring m a unit of work for the atomic set of f and g in n.

Preventing Global Locks. Without further adjustments, inferring aliases this way can lead to undesirable global locks, as shown in Figure 6.2: if an alias merges an atomic set in a thread object with an atomic set S in an object shared between threads, then the thread's methods become units of work for S. Consequently, only one thread object can execute at a time, making (this part of) the program sequential.

We apply the following heuristic to detect this situation and lower the respective alias beliefs. Whenever a thread t accesses a field in object o, we record t as the owner of o. Using this data, we maintain an *alias factor* α for objects. Consider the situation in Figure 6.2, just after the left thread t_l's call to `getNextURL()` has returned. At this point, t_l owns itself, the manager object, and the list object. When the right thread t_r accesses its local `manager` field just after that, BAIT detects that t_r owns the object that contains the accessed `manager` field (itself), but another thread owns the object that is the field's *value* (t_l owns the manager object). Therefore, the thread object t_r and the manager object appear to belong to two different clusters in the object graph upon which different threads operate concurrently. Merging these clusters with an alias would remove the concurrency. Therefore, we set a fixed alias factor α in the range $(0, 1)$ for the manager object (the field's value). Otherwise, if t_r was the owner of itself *and* the manager object, we set α based on the recorded (same-thread) distance between the accesses, which can result both in lowering or raising belief in an alias.

Given an atomic field access event, we use the computed alias factor α for the field-containing object as weight when updating an alias affinity matrix. Adapting Equation 6.3, the updated affinity matrix A'_n for the name n of o is thus computed as

$$A'_n = A_n\big[(f,g) \mapsto \alpha \cdot \ell(d,a) \cdot A_n(f,g)\big].$$

In the example shown in Figure 6.2, the alias factor $\alpha < 1$ for the manager object prevents the small distance between the observed accesses of `manager.urls` $(6,7$ and $8,9)$ in the `run()` methods from increasing the odds of the problematic alias from `DownloadThread.manager` to `DownloadManager.urls`.

A slight modification of the heuristic is necessary to account for clusters consisting of more than two objects. In its current form, the heuristic detects a different owner thread for the first accessed object o of a cluster, and the same owner for the second object v, say, accessed via field f in o. However, the access of f establishes the current thread as the owner of o. Thus, when accessing a third object w via the field g in o, the heuristic would detect different owners again, discouraging an alias even though the previous thread operated on o, v, and w. BAIT solves this problem by not only recording the current owning thread t_o for each object, but also the previous (distinct) owning thread t'_o. Different clusters are detected only if $t_o \neq t_v$ and $t'_o \neq t_v$. Thus, for the access of w we have $t'_o = t_w$ and correctly associate w with o and v.

6.4.7 Atomic Set, Alias and Unit of Work Formation

After the workload has finished executing, all atomic set field affinity matrices are merged into a single matrix. From this combined matrix, the atomic sets are extracted by using the matrix values as edge-weights on the fully-connected graph of all fields (node set Fd), removing the edges with weight less than a threshold (we use 1), and grouping the fields in the remaining connected components by their declaring class (accounting for inheritance). The atomic sets are added as annotations to the class hierarchy, which forms the basis for computing aliases using the alias affinity matrices. Finally, units of work are inferred using the class hierarchy and the alias affinity matrices.

6.5 Implementation

We have implemented BAIT in a tool chain for Java programs[1]. The tool chain consists of a Java byte code instrumenter and an inference tool. The instrumenter uses WALA's[2] Shrike library to insert calls to the field access tracing library into the input byte code. After instrumentation, the target program must be executed to generate field access traces, and simultaneously infer affinity matrices.

[1]To be released at `http://osl.cs.illinois.edu/software/`

[2]T. J. Watson Libraries for Analysis. `http://wala.sf.net`

For reasons of simplicity and performance, our implementation uses piecewise linear approximations of the logistic functions for mapping distances to likelihood ratios for atomic sets and aliases. To allow the tool to process realistic programs, we have extended it with the ability to handle arrays, synchronized blocks, and wait–notify synchronization, as explained in earlier work [61]. In addition, we have included heuristics that add special handling of monitor variables and constructors, and removal of non-aliased `final` fields from atomic sets. While the algorithm requires tracking all names that a field-owning object can have, the implementation only tracks the last known name at runtime. We believe this gives a reasonable tradeoff between overhead and correctness.

The tool ignores the limitations of the current AJ implementation. Consequently, it does not suggest required refactorings such as making nested classes into top-level classes, adding getter and setter methods, and using only one atomic set per class.

The implementation has several parameters that can dramatically affect the inferred annotations, most prominently the parameters that define the piecewise approximations of the logistic functions for atomic set and alias likelihoods. We have calibrated the parameters using mainly small test cases where the desired results are easy to determine. A method for automatically tuning parameters for specific codebases, e.g., based on the differences in output between several runs of the same workload, should be possible to develop, but we leave this as future work.

6.6 Evaluation

We measure the performance of Bait by the quality of the inferred annotations. The highlights of our results are presented in this section.

6.6.1 Approach

We evaluate Bait by running our implementation on all but one of the Java programs for which an AJ version is publicly available, and comparing the inferred annotations to the manually added annotations. We follow this subjective *qualitative* approach for two reasons. First, the goal of the algorithm is to infer annotations that not only enforce, but also document the intended concurrency semantics of the program. Evaluating how well the inferred annotations meet this goal requires manual inspection of the code. Simple quantification of the differences between manual and inferred annotations alone—for example their number or size—does not convey meaningful information because most AJ versions have been refactored and structurally differ from the Java versions; furthermore, some of the manual annotations are incomplete and sometimes even incorrect. Second, using other quantitative measures like execution speed is infeasible because the prototype AJ compiler is currently defunct.

6.6.2 Program Corpus

Table 6.1 lists the programs used to evaluate the algorithm. The list includes all Java programs for which an AJ version is publicly available, except *cewolf*. The *cewolf* library was excluded because it contained too few AJ annotations to justify the effort of creating a fuzzing tool for it. For every program except *collections*, the corpus also includes the compiled AJ version. Both versions are used in the evaluation. These Java programs were manually converted to AJ by Dolby et al. [62]. Archives containing the source code of the conversions are available on the *Data-Centric Concurrency Control* project website[3]. The AJ variant of the Java collections framework was kindly provided by Frank Tip.

6.6.3 Method

Each program in the corpus is first instrumented and then run three times using the same workload. For *elevator* and *tsp2*, the workload consists of example input files distributed with the programs; *weblech* is used to aggregate files from a local web server; the *collections* and *jcurzez* libraries are used for random operations by custom fuzzing programs. Creating the fuzzing programs took about half a day per library. However, this effort could be automated, or unit tests could be used instead where available. All workloads were set large enough to trigger the use of multiple operating system threads by the JVM in order to obtain observations of fine-grained interleavings. Comparing the annotations inferred for three separate runs gives us insight into the effects of (random) thread scheduling and allows us to verify that the annotations likely reflect consistent program behavior. Consequently, before analyzing the results, we remove spurious annotations and consolidate the remaining annotations from all runs. We use the same calibrated parameters for all programs except *collections*, where we adjust the approximated logistic function for distance-to-likelihood mapping to increase the odds of inferring aliases. This is necessary to accommodate deficiencies in the *collections* fuzzer.

Next, we compare these inferred annotations against the ones Dolby et al. manually inserted when converting the programs to AJ. For every difference, we investigate whether it results in disparate program behavior by analyzing the source code of both variants. Furthermore, we discuss the root cause that led to inferring a differing annotation.

Several refactorings were applied during the development of the AJ variants to meet the requirements of AJ, to work around limitations of the used implementation, and to simplify the conversion. Refactorings to meet requirements include introducing getter and setter methods for fields. Workaround refactorings include flattening nested classes and splitting classes to achieve concurrent execution. Simplifications include dropping specialized iterator classes in the *collections* framework.

[3]`http://sss.cs.purdue.edu/projects/aj/`

Table 6.1: Programs used to evaluate the algorithm. The *kLoC* column lists the number of thousand lines of source code in the Java version of the program, excluding comments and empty lines. The *Classes* column shows the number of classes in the program. In parentheses follows the number of classes that contain at least one manually added AJ annotation (`atomicset`, `atomic`, or `unitfor`).

Program	*Description*	*kLoC*	*Classes*
collections	OpenJDK 1.6 collections	11.1	171 (43)
elevator	Elevator simulation	0.3	6 (2)
jcurzez1	UI library (low concur.)	2.7	78 (9)
jcurzez2	UI library (high concur.)	2.8	79 (6)
tsp2	Traveling salesman	0.5	6 (2)
weblech	Web site mirror tool	1.3	12 (2)

We do not report the processing times because in a source code conversion workflow, it suffices to execute the tool once. We simply note that for each program, instrumentation finished within seconds; collecting the observations and inferring the annotations generally took a few seconds, and never more than one minute, on an Intel Core i5 processor with 2 GB of RAM.

6.6.4 Results

The inferred annotations can differ from the manual annotations in both missed and added atomic set, alias, and unit of work definitions. The consequences vary:

- The most critical kind of difference is a missing or incomplete atomic set, implying that some fields that were intended to be protected from interleaved accesses remain unprotected, which may result in a race condition. Additional atomic sets can lead to deadlock, but this is not a severe problem because deadlock caused by atomic sets can be statically recognized [134].
- Missing aliases result in synchronization overhead and can lead to high-level races. Additional aliases reduce the potential for concurrency in the program. However, extraneous aliases cannot lead to errors like race conditions or deadlocks.
- Missing unit of work declarations can lead to race conditions. Additional declarations may reduce the concurrency in the program and lead to (statically recognizable) deadlock.

In the comparisons, we ignore secondary causes of differences that either reflect refactorings, or can be fixed through a better tool implementation or workload. Overall, then, the inferred annotations mostly agree with the manual annotations. We discuss the compared programs in turn.

collections Most atomic sets inferred by Bait are complete, but some sets are lacking some manually added fields. In one case, this omission is clearly motivated, since the inclusion introduces a global lock, highlighting a mistake in the manual annotations. Several abstract classes are missing units of work, which in most cases are instead inferred for their implementing classes. In some other cases, the omission of units of work are attributable to the fuzzer, which is clearly incomplete. For example, the fuzzer does not access iterators from different threads.

elevator Bait does not infer the manually annotated units of work, but as these manual declarations are formulated, they are incompatible with the AJ specification. The tool infers several additional atomic sets that serve as documentation of existing behavior. The tool omits one alias due to workload issues, and adds two aliases from arrays to their elements that document existing behavior.

jcurzez1 Bait correctly infers all manually added atomic sets for all but one class, where one set is missing a field due to a conservative choice of alias inference parameters. One new atomic set is introduced that prevents races that can occur in the workload, while several others are added that document existing behavior. Some manual units of work annotations are not inferred due to incomplete workloads or tool deficiencies that are straightforward to fix. One inferred unit of work prevents inadvertent races and therefore highlights a bug in the manual annotations. One correct additional alias is inferred and one manual alias is omitted. This omission is due to interleaved access, and shows that the original synchronization in the program is incomplete.

jcurzez2 Bait again correctly infers all manually added atomic sets for all but one class, where a set is again missing a field due to alias inference parameters, and adds more sets that document behavior. Alias inference follows the pattern from *jcurzez1*. Some manually added unit of work annotations are not inferred, but this is either due to insufficient workload or minor tool deficiencies. Two cases of inferred added units of work indicate omissions in the manual annotations that can lead to races.

tsp2 Bait infers all manual annotations, while correctly adding several atomic sets and aliases that document existing behavior.

weblech Bait infers all manual annotations, while adding several sets and aliases.

Inferring annotations for the manually ported AJ variants of the programs yields results similar to the those of the original variants. This indicates that the manual annotations capture most of the original program's behavior. It also shows that the manual refactorings do not influence the inference to any significant extent.

The effects of the inferred annotations on the behavior of *collections*, the *jcurzez* variants, and *tsp2* match the effects of the manual annotations.

In *elevator*, the threads synchronize using the elements of a globally shared array `floors` as monitors. The developers of AJ use a generalized **`unitfor`** annotation to circumvent global locking. However, while having the right effects in the AJ implementation, this annotation violates the atomic set typing rules because it contains non-final segments in its atomic set designator. Our algorithm does not include the array `floor` in any atomic set, and hence races on the array are not prevented.

The annotations inferred for *weblech* impose a global lock: the download threads in *weblech* execute a single shared `Runnable` object; adding an atomic set to this object effectively prevents any concurrent execution, that is, downloading. A solution to this problem is to split the `Runnable` object. The manually ported AJ variant of *weblech* follows this approach, but the refactoring leaves the crucial blocking network access inside a unit of work for the atomic set of `Runnable`, and thereby fails to enable concurrent downloading. Thus, the annotations for *weblech* underline for argument for tool support for suggesting program refactorings.

BAIT's inferred annotations for *jcurzez1* reveal race conditions in the classes `Cell`, `Cursor`, and `Pen`. In its AJ variant, the racing fields of `Cursor` and `Pen` are protected by an atomic set. Since both the library's control- and data-centric synchronization was added by the AJ developers, this documents that the race is unintended, which underlines the difficulty of defining control-centric synchronization. The malign race in class `Cell` and the lack of a manual atomic set definition for this class are proof that understanding the concurrency structure of other people's programs is hard, which supports our case for automating the necessary reasoning. The mistakenly added atomic set in *collections* adds further support.

6.7 Case Studies

We investigate the extent to which BAIT can deal with rare races—likely the result of programming errors—and still infer correct annotations, by running a test program with intermittent unsynchronized field access with different parameters. For the program, an atomic set is consistently inferred as long as unsynchronized accesses are about one tenth as many as the atomic accesses, or less. In addition, we explore the feasibility of our approach on large-scale software by conducting annotation inference case studies on the widely-used programs *Lucene*[4] and *Xalan*[5], with workloads from the DaCapo benchmark suite [22], release *9.12-bach*. The program details are listed in Table 6.2. These two case studies show that the tool can be used on large programs to derive annotations that clarify control-centric synchronization behavior and prevent the occurrence of rare races in future development. Annotations also identify and document clusters of classes related through aliases between their atomic sets.

[4]`http://lucene.apache.org`

[5]`http://xml.apache.org/xalan-j/`

Table 6.2: Programs used in case studies of the inference algorithm. The *kLoC* column lists the number of thousand lines of source code in the Java version of the program, excluding comments and empty lines. The *Classes* column shows the number of classes in the program.

Program	*Version*	*Description*	*kLoC*	*Classes*
Lucene	2.4.1	Text search	68.3	559
Xalan	2.7.1	XSLT processor	172.3	1514

6.7.1 Inference Robustness Study

To test BAIT's robustness, we devised a program where two threads repeatedly set the `firstName` and `lastName` fields of a shared object which is an instance of a class `Person`. Threads either set the two fields atomically using the **`synchronized`** method `setNames(String,String)`, but sometimes use the unsynchronized method `setLastName(String)` to only set the last name. Threads do this by repeatedly calling a method `doWork(int)`, shown in Listing 6.3, while incrementing the argument `iteration`. When unsynchronized access only occurs every tenth iteration, as in the figure, an atomic set with `firstName` and `lastName` for `Person` is correctly inferred across runs. However, when unsynchronized access occurs every fifth call, the atomic set is sometimes inferred and sometimes not inferred; the outcome is likely determined by the scheduler. With every third call unsynchronized, the atomic set is not inferred at all.

```
protected void doWork(int iteration) {
  long tId = Thread.currentThread().getId();
  if (iteration % 10 == 0) { // Non-atomic access
    sharedPerson.setLastName("Last" + tId);
    sharedPerson.setLastName("Last" + tId);
  } else { // Atomic access
    sharedPerson.setNames("First" + tId, "Last" + tId);
  }
}
```

Listing 6.3: Snippet from program with intermittently interleaved field accesses

This program is an extreme case in terms of intermittent access—for actual programs, access distance is more likely to be the determining factor of atomic set inference. The results nevertheless demonstrate that a degree of robustness against intermittent incorrect program behavior is inherent in BAIT.

6.7.2 *Lucene*

We use the *lusearch* DaCapo benchmark as workload for *Lucene*. This benchmark does text search of keywords, using several threads, over a corpus of data comprising the works of Shakespeare and the King James Bible. Hence, BAIT produces many annotations for the parts of *Lucene* that deal with text search using this workload, but few for the parts that deal with mutating search indexes.

Notably, the algorithm infers an atomic set containing most fields of the class `SegmentReader` in the package `org.apache.lucene.index`. The atomic set includes the field `deletedDocs`, which is accessed in the methods shown in Listing 6.4. The method `isDeleted()` is **`synchronized`**, which suggests that `deletedDocs` is accessed by concurrent threads. However, neither the method `doDelete(int)`, which writes `deletedDocs`, nor the method `numDocs()`, which reads it, are **`synchronized`**. Clearly, there is at present a potential for races involving `deletedDocs` when using these methods; for example, when a thread executing `doDelete(int)` sets `deletedDocs` to a non-**`null`** value while other threads simultaneously run `isDeleted()` or `numDocs()`. Adding the field to the atomic set rules out such races.

```
protected void doDelete(int docNum) {
  if (deletedDocs == null)
    deletedDocs = new BitVector(maxDoc());
  deletedDocsDirty = true;
  undeleteAll = false;
  if (!deletedDocs.getAndSet(docNum))
    pendingDeleteCount++;
}
/*...*/
public synchronized boolean isDeleted(int n) {
  return (deletedDocs != null && deletedDocs.get(n));
}
/*...*/
public int numDocs() {
  int n = maxDoc();
  if (deletedDocs != null)
    n -= deletedDocs.count();
  return n;
}
```

Listing 6.4: Snippet from class `SegmentReader` in *Lucene*

BAIT correctly adds annotations to make instances of priority queues for text search scores thread safe, namely, the class `HitQueue` in `org.apache.lucene.search`. First, in the abstract class `PriorityQueue` which `HitQueue` extends, an atomic set with the fields `heap`, `maxSize`, and `size` is added, with an alias from the array field `heap` to its elements and to the atomic set of class `ScoreDoc`, whose instances are stored in the queue. In `HitQueue`, the method `lessThan` is added as unit of work for the atomic sets of both its `ScoreDoc` arguments.

The atomic set for `CompoundFileReader` in the package `org.apache.lucene.index` includes the fields `entries` and `stream`, with an alias from the latter field to the atomic set of `FSDirectory.FSIndexInput` in `org.apache.lucene.store`. On inspection of the class `CompoundFileReader`, the methods `close()` and `openInput()`, which both access `entries` and `stream`, are **`synchronized`**, lending support that there is an invariant between the fields and justifying their inclusion in the atomic set.

6.7.3 *Xalan*

Xalan is an XSLT processor for transforming XML documents. In the DaCapo benchmark, *Xalan* repeatedly transforms a set of XML documents using several threads, scaled to the available processor cores.

The atomic sets derived by BAIT often have many members for classes with many fields. For example, class `TransformerImpl` in `org.apache.xalan.transformer` is annotated with a set with 34 fields. Closer inspection of the class suggests that only a few of these fields are accessed by concurrent threads, as evidenced by the use of locks, and that some of the fields can only be set once, for instance in the constructor. While it could be justified to have large atomic sets in some cases, the results suggest it is worthwhile enhancing our tool using static analyses, e.g., for determining field mutability, to reduce the number of fields in atomic sets.

For the class `ElemTextLiteral` in `org.apache.xalan.templates`, the algorithm infers an atomic set containing the **`char`** array field `m_ch`, and the `String` field `m_str`, among others. The **`synchronized`** method `getNodeValue()` in the class, shown in Listing 6.5, suggests that `m_str` is accessed by concurrent threads, and that `m_str` is related to `m_ch` in that the former is the string representation of the latter. However, the presence of a **`public`** mutator method `setChars(`**`char`**`[])` for `m_ch`, also shown in the figure, makes it trivial to break the invariant by calling the method with a new char array after a call to `getNodeValue()`, and enables races between calls by different threads to the two methods.

As far as we have been able to determine, the method `setChars(`**`char`**`[])` is actually only called in a method which creates and initializes a `ElemTextLiteral` instance, before that instance is passed as a method argument and potentially used by other threads. Still, the inclusion of `m_ch` and `m_str` in the atomic set makes the invariant between them more explicit, and protects from races that could otherwise be introduced by future development.

When executing the workload, BAIT reports races on the fields `m_hasTextLitOnly` and `m_prefixTable` in the class `ElemTemplateElement` in `org.apache.xalan.templates`. The inferred atomic set for the class include both of these fields, ruling out such races.

6.8 Discussion

BAIT successfully infers annotations for atomic sets, aliases, and units of work from the execution traces of a program; if these are not available they have to be

```
public void setChars(char[] v) {
  m_ch = v;
}
/*...*/
public synchronized String getNodeValue() {
  if(null == m_str) {
    m_str = new String(m_ch);
  }
  return m_str;
}
```

Listing 6.5: Snippet from class `ElemTextLiteral` in *Xalan*

generated by executing the program. In particular, converting isolated modules of a large code base requires unit tests which execute these modules. The algorithm further assumes that all observed execution traces are mostly correct, that is, reflect programmer intent. This assumption can hold even if a program contains bugs: schedule-dependent *Heisenbugs* that never (or rarely) appear during testing will still produce odds in favor of an invariant. In this case, the inferred annotations will prevent the bugs in future executions.

The major factor driving the suggestion of additional annotations is the documentation of *obvious* behavior. Obvious behavior concerns high-level understanding of a program's concurrency semantics. Developers use this understanding to avoid annotating classes they deem irrelevant for achieving the intended behavior. The inference algorithm lacks this concept of obviousness and generates annotations for all classes. From the perspective of project-external developers, these annotations provide a guard against accidentally violating behavior invariants, while at the same time documenting these invariants.

The degree of concurrency in a program with inferred annotations depends on the concurrency manifest in the execution traces that are used. It is therefore important to collect traces using workloads that trigger as much correct concurrent behavior as possible. It would be feasible to automate the generation of workloads, for example using concolic execution to explore thread scheduling.

Another factor limiting concurrency is the current lock-based implementation of atomic sets. Our algorithm treats read–read sharing of fields as non-interleaved access and therefore includes these fields in atomic sets. In the converted program, the fields are therefore protected by locks, preventing the concurrent reading observed in the execution traces. Although unnecessarily restrictive, the resulting behavior will be correct. Better implementations of atomic sets, for example using software transactional memory, or inferring advanced annotations such as **`fastread`**, partial **`unitfor`**, and **`internal`**, would improve the degree of concurrency. As demonstrated by Dolby et al. [62], these annotations may have a dramatic effect on a program's performance.

Our evaluation focuses on whether BAIT produces correct, reasonable, and useful annotations. Another important aspect is whether programs that have been annotated by Bait can be executed with acceptable overhead when concurrency control is based on those annotations. We leave such a performance study as future work, since it would require the development of a fast, high-quality AJ compiler, to replace the proprietary, defunct prototype compiler. Nevertheless, based on previous results when compiling annotated programs from the SPECjbb benchmark, we expect that throughput between 80 % and 90 % can be achieved, depending on the degree of optimization [62]. In addition to such a performance analysis, there is a wide range of options open for enhancing our tool with heuristics to improve annotation quality, such as performing escape analysis to detect thread-local fields and use of statically derived information on classes with immutable instances.

6.9 Related Work

The automatic inference of a program's concurrency semantics has been treated in the context of data race detection. There, the concurrency semantics is used to warn about violations of the likely *intended* atomicity semantics of variables.

A dynamic approach that learns the atomicity intentions for shared variables from execution traces is the *AVIO* system of Lu et al. [130, 129]. *AVIO* observes the read and write operations on a shared variable and treats it as atomic if all operations were serializable. Observing each variable in isolation, *AVIO* can only detect low-level data races. In contrast, Artho et al. [12] introduce the notion of high-level data races and explicitly design their dynamic algorithm to consider races on sets of semantically related variables. The *AssetFuzzer* algorithm of Lai et al. [114] uses partial order relaxation to detect potential, but unmanifested, violations in the execution trace. All of these methods are similar to our algorithm in that they work without user annotations. The *Atomizer* system of Flanagan and Freund [73] additionally considers *windows of vulnerability*, but requires a few source code annotations and potentially raises false alarms.

The *MUVI* tool of Lu et al. [127] follows a static approach to inferring atomicity intentions. It computes variable correlations by mining the program source code. As opposed to our tool, it relies on the static (source code) distance to infer semantic relationships.

The static heuristic [84, 191] of defining one atomic set per class that contains all non-static fields has also been proposed in the context of race detection. Targeting race detection, none of the aforementioned approaches considers aliasing information, which is essential for our use case.

Huang and Milanova propose a static inference system for AJ types that significantly reduces the number of annotations that a developer has to write [96]. While simplifying the use of AJ, it needs a set of foundational annotations. Hence, their and our methods complement each other: the static inference rules propagate the base annotations inferred by our analysis, yielding a complete set of AJ annotations.

Liu et al. [125] describe a technique for statically inferring atomic sets based on program dependence analysis. The inferred sets are then used for finding atomic composition bugs dynamically in programs. This is a different focus compared to our algorithm, whose main aim is to provide annotations for documentation and safe execution. In addition, our algorithm also infers aliases, which are arguably harder to infer than atomic sets, least of all statically.

Flanagan et al. [75] present a sound and complete dynamic atomicity checker for Java programs. The tool, Velodrome, takes a workload and list of methods that are assumed to be atomic as input, and outputs a list of atomicity violations. Biswas et al. [21] improve on the significant overhead introduced by Velodrome in their DoubleChecker tool, while maintaining soundness and completeness. A tentative list of atomic methods can be derived from the annotations produced by our tool by simply enumerating all methods that are units of work for some atomic set. When used as input to an atomicity checker, the resulting analysis is sound but not complete with respect to AJ semantics, since aliasing introduces requirements on cross-method atomicity. If an atomicity checker is extended to support the fine-grained atomicity specifications offered by aliases, the resulting tool chain could be used for automatically identifying more subtle atomicity violations than are possible at present.

Atomic sets take a declarative approach to synchronization. *Synchronizers* [79, 59] provide a similar notion in the context of Actor systems, where they constrain the message dispatch in a group of actors. The available constraints differ from atomic sets in that Synchronizers can provide *temporal* atomicity—messages arrive at the same time—not the *spatial* atomicity offered by atomic sets. Furthermore, Synchronizers cannot easily express the non-interleaving of message sequences, which is the Actor equivalent of non-interleaved access to shared data, and do not support transitive extensions similar to aliases in atomic sets.

By boosting belief in the existence of an invariant after atomic access and maintaining or possibly even strengthening that belief unless witnessing interleaved access, Bait follows the approach of *accentuating the positive* [203, 129] by suppressing rarely observed Heisenbugs that violate atomicity. A study of real-world concurrency bugs [128] finds that nearly half of all errors are related to atomicity; with deadlocks ruled out, that fraction rises to nearly 70 %. While this kind of safety comes at the cost of a coarser concurrency semantics, the experiments of Weeratunge et al. [203] suggest that a low runtime overhead of 15 % can be achieved.

6.10 Conclusions

We presented a novel algorithm, Bait, that dynamically infers annotations for data-centric synchronization in multi-threaded programs that use control-centric synchronization. The algorithm uses Bayesian probabilistic inference to incorporate evidence from program execution traces into overall odds of the existence of invariants between fields of objects. By taking into account locality properties (distance) of field accesses in evidence, improving accuracy with more data from traces, bounding observation data by code base size, and incorporating a heuristic to prevent inferring aliases that introduce global locks, the algorithm improves on the state of the art for dynamic inference of data-centric synchronization annotations.

Our evaluation of Bait compares manual annotations of available AJ programs to the annotations produced by our Java-based algorithm implementation. The results show that most manual annotations are inferred properly, while highlighting several bugs in the manual annotations and in the original programs. These bugs lend support to our premise that while understanding program concurrency semantics is a major obstacle for manually adding data-centric annotations to legacy code, there are considerable benefits of having such annotations. Our case studies reveal that many useful annotations can be derived by the algorithm implementation for large, widely-used programs, and thus potentially used to migrate these programs away from a control-centric style of concurrency management to the safer data-centric style.

Acknowledgements

The authors would like to thank Minas Charalambides, Amarin Phaosawasdi, and Tarun Prabhu for their insightful comments on earlier versions of this paper. This publication was made possible in part by sponsorships from the Army Research Office under award W911NF-09-1-0273, as well as the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement FA8750-11-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Bibliography

[1] 4WARD project website. URL `http://www.4ward-project.eu`. Retrieved 2014-09-01.

[2] Daniel J Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, 2009.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[4] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, January 1997.

[5] Philip E. Agre. P2P and the promise of Internet equality. *Communications of the ACM*, 46(2):39–42, February 2003.

[6] Akka project website. URL `http://akka.io`. Retrieved 2014-08-24.

[7] Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 353–368. Springer, Berlin, Germany, 2011.

[8] Amazon Web Services website. URL `https://aws.amazon.com`. Retrieved 2014-08-25.

[9] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[10] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4):34–41, April 2005.

[11] Ken Arnold. *The Jini Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[12] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[13] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[14] Baruch Awerbuch and David Peleg. Online tracking of mobile users. *Journal of the ACM*, 42(5):1021–1058, September 1995.

[15] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, September 1999.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles*, SOSP '03, pages 164–177. ACM, 2003.

[17] Petra Berenbrink, Tom Friedetzky, Leslie Ann Goldberg, Paul Goldberg, Zengjian Hu, and Russell Martin. Distributed selfish load balancing. *SIAM Journal on Computing*, 37(4):1163–1181, 2007.

[18] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 81–94. ACM, 1990.

[19] Lorenzo Bettini, Viviana Bono, Rocco Nicola, Gianluigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. In Corrado Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, Berlin, Germany, 2003.

[20] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, pages 265–276. ACM, 2005.

[21] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. Doublechecker: Efficient sound and precise atomicity checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 28–39. ACM, 2014.

[22] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump,

Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190. ACM, 2006.

[23] Joshua Bloch. Extra, extra – read all about it: Nearly all binary searches and mergesorts are broken, June 2006. URL `http://googleresearch.blogspot.se/2006/06/extra-extra-read-all-about-it-nearly.html`. Retrieved 2014-08-13.

[24] Jeff Bonwick. Rampant layering violation?, 2007. URL `http://blogs.oracle.com/bonwick/entry/rampant_layering_violation`. Retrieved 2014-02-18.

[25] Michael Burrows and K. Rustan M. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience*, 16(12):1161–1172, 2004.

[26] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer, Berlin, Germany, second edition, 2011.

[27] Matthew Caesar, Tyson Condie, Jayanthkumar Kannan, Karthik Lakshminarayanan, and Ion Stoica. ROFL: routing on flat labels. *ACM SIGCOMM Computer Communication Review*, 36(4):363–374, August 2006.

[28] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, Berlin, Germany, 1998.

[29] Richard Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, 2001.

[30] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[31] Denis Caromel and Ludovic Henrio. *A theory of distributed objects - asynchrony, mobility, groups, components.* Springer, Berlin, Germany, 2005.

[32] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 123–134. ACM, 2004.

[33] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459–495, April 2009.

[34] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdag, Robert Heaphy, and Lee Ann Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Parallel and Distributed Processing Symposium, 2007. IEEE International*, IPDPS 2007, pages 1–11. IEEE, 2007.

[35] Vinton G. Cerf, Yogen Dalal, and Carl Sunshine. Specification of Internet Transmission Control Program. RFC 675, December 1974. URL `http://www.ietf.org/rfc/rfc675.txt`.

[36] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[37] Marsha Chechik and Andre Wong. Formal methods when money is tight. In *1st Workshop on Economics-Driven Software Engineering Research*, EDSER, May 1999.

[38] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[39] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[40] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[41] David Cole. The data center management "elephant". White paper, PTS Data Center Solutions, 2010. URL `http://www.ptsdcs.com/wpp/PTS/The%20Infrastructure%20Management%20Elephant.pdf`.

[42] Sylvain Conchon and Fabrice Le Fessant. Jocaml: mobile agents for Objective-Caml. In *Agent Systems and Applications, 1999 and Third International Symposium on Mobile Agents. Proceedings. First International Symposium on*, pages 22–29. IEEE, 1999.

[43] The Coq Proof Assistant website. URL `http://coq.inria.fr`. Retrieved 2014-08-13.

[44] Alejandro Cornejo, Fabian Kuhn, Ruy Ley-Wild, and Nancy Lynch. Keeping mobile robot swarms connected. In *Proceedings of the 23rd International Conference on Distributed Computing*, DISC '09, pages 496–511. Springer, 2009.

[45] Alejandro Cornejo and Nancy Lynch. Fault-tolerance through k-connectivity. In *IEEE International Conference on Robotics and Automation: Workshop on Network Science and Systems Issues in Multi-Robot Autonomy*, 2010.

[46] Alejandro Cornejo and Nancy Lynch. Reliably detecting connectivity using local graph traits. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems*, volume 6490 of *Lecture Notes in Computer Science*, pages 87–102. Springer, Berlin, Germany, 2010.

[47] Oracle Corporation. Java Remote Method Invocation API (Java RMI), 2014. URL `http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/`.

[48] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, and Vittorio Scarano. Distributed load balancing for parallel agent-based simulations. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 62–69. IEEE, 2011.

[49] Leslie Daigle. Prediction: IPv6 adoption hits double digits in 2014, January 2014. URL `http://internetsociety.org/blog/tech-matters/2014/01/prediction-ipv6-adoption-hits-double-digits-2014`. Retrieved 2014-08-24.

[50] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 223–234. ACM, 2013.

[51] Mads Dam and Karl Palmskog. Efficient and fully abstract routing of futures in object network overlays. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 49–60. ACM, 2013.

[52] Mads Dam and Karl Palmskog. Location independent routing in process network overlays. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 715–724. IEEE, Feb 2014.

[53] Mads Dam and Karl Palmskog. Location independent routing in process network overlays. *Service Oriented Computing and Applications*, 2014. To appear.

[54] Mads Dam and Rolf Stadler. A generic protocol for network state aggregation. In *Proceedings of Radiovetenskap och Kommunikation*, RVK, 2005.

[55] Christian Dannewitz, Dirk Kutscher, Börje Ohlman, Stephen Farrell, Bengt Ahlgren, and Holger Karl. Network of information (netinf) - an information-centric networking architecture. *Computer Communications*, 36(7):721–735, April 2013.

[56] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, Berlin, Germany, 2007.

[57] Michael J. Demmer and Maurice P. Herlihy. The arrow distributed directory protocol. In Shay Kutten, editor, *Distributed Computing*, volume 1499 of *Lecture Notes in Computer Science*, pages 119–133. Springer, Berlin, Germany, 1998.

[58] Yuxin Deng and Jean-Francois Monin. Verifying self-stabilizing population protocols with coq. In *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*, pages 201–208. IEEE, July 2009.

[59] Peter Dinges and Gul Agha. Scoped synchronization constraints for large scale actor systems. In Marjan Sirjani, editor, *Coordination Models and Languages*, volume 7274 of *Lecture Notes in Computer Science*, pages 89–103. Springer, Berlin, Germany, 2012.

[60] Peter Dinges, Minas Charalambides, and Gul Agha. Automated inference of atomic sets for safe concurrent execution. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 1–8, New York, NY, USA, 2013. ACM.

[61] Peter Dinges, Minas Charalambides, and Gul Agha. Automated inference of atomic sets for safe concurrent execution. Technical report, University of Illinois at Urbana-Champaign, April 2013. URL `http://hdl.handle.net/2142/43357`.

[62] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems*, 34(1):4:1–4:48, May 2012.

[63] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software: Practice and Experience*, 21(8):757–785, 1991.

[64] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[65] Erlang programming language website. URL http://www.erlang.org. Retrieved 2014-08-24.

[66] Erlang/OTP 17. URL http://www.erlang.org/doc/. Retrieved 2014-08-24.

[67] Dave Evans. The Internet of Things: How the next evolution of the internet is changing everything. White paper, Cisco, Apr 2011. URL http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

[68] Eyal Even-Dar and Yishay Mansour. Fast convergence of selfish rerouting. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 772–781. Society for Industrial and Applied Mathematics, 2005.

[69] John Field and Carlos A. Varela. Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 195–208. ACM, 2005.

[70] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2): 374–382, April 1985.

[71] Simon Fischer and Berthold Vöcking. Adaptive routing with stale information. *Theoretical Computer Science*, 410(36):3357–3371, 2009.

[72] Cormac Flanagan and Mattias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, January 1999.

[73] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71 (2):89–109, 2008.

[74] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133. ACM, 2009.

[75] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 293–303. ACM, 2008.

[76] Cédric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer, Berlin, Germany, 1996.

[77] FP7-231620 (HATS) Project. Deliverable 1.2: Full ABS modeling framework, March 2011. URL `http://www.hats-project.eu/sites/default/files/Deliverable12.pdf`.

[78] Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failures. In Martin Abadi and Luca Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 368–382. Springer, Berlin, Germany, 2005.

[79] Svend Frølund and Gul Agha. A language framework for multi-object coordination. In Oscar M. Nierstrasz, editor, *ECOOP '93 – Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 346–360. Springer, Berlin, Germany, 1993.

[80] David Geer. Chip makers turn to multicore processors. *IEEE Computer*, 38 (5):11–13, May 2005.

[81] Google App Engine website. URL `https://developers.google.com/appengine/`. Retrieved 2014-08-25.

[82] J. N. Gray. Notes on data base operating systems. In M. J. Flynn, editor, *Operating Systems: An Advanced Course*. Elsevier, Amsterdam, Netherlands, 1978.

[83] James Hamilton. Architecture for modular data centers. In *Conference on Innovative Data Systems Research*, 2007.

[84] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 231–240. ACM, 2008.

[85] Robert Harper. Parallelism is not concurrency, March 2011. URL `https://existentialtype.wordpress.com/2011/03/17/parallelism-is-not-concurrency/`. Retrieved 2014-08-13.

[86] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2013.

[87] Dragan Havelka, Christian Schulte, Per Brand, and Seif Haridi. Thread-based mobility in Oz. In Peter Roy, editor, *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *Lecture Notes in Computer Science*, pages 137–148. Springer, Berlin, Germany, 2005.

[88] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-threaded active objects. In Rocco De Nicola and Christine Julien, editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 90–104. Springer, Berlin, Germany, 2013.

[89] Ludovic Henrio and Muhammad Uzair Khan. Asynchronous components with futures: Semantics and proofs in Isabelle/HOL. In *Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures*, volume 264 of *Electronic Notes in Theoretical Computer Science*. Elsevier, August 2010.

[90] Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, and Eugenio Zimeo. First class futures: Specification and implementation of update strategies. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 295–303. Springer, Berlin, Germany, 2011.

[91] Charles Herzfeld. Charles Herzfeld on ARPANET and computers. URL `http://inventors.about.com/library/inventors/bl_Charles_Herzfeld.htm`. Retrieved 2014-07-15.

[92] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[93] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 41–52. ACM, 2002.

[94] C. Anthony R. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, 1983.

[95] Gerard J. Holzmann. *The Spin Model Checker.* Addison-Wesley Professional, Boston, MA, USA, 2003.

[96] Wei Huang and Ana Milanova. Inferring AJ types for concurrent libraries. In *Workshop on the Foundations of Object-Oriented Languages at OOPSLA*, FOOL 2012, pages 82–88, 2012.

[97] Isabelle website. URL `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`. Retrieved 2014-08-13.

[98] ISO/IEC 14882:2012(E). Programming language c++. International standard, ISO, Geneva, Switzerland, 2012. Section 30: Thread support library.

[99] ISO/IEC 7498-1. Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. International Standard, ISO, Geneva, Switzerland, 1994.

[100] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT '09, pages 1–12. ACM, 2009.

[101] Alan Jeffrey and Julian Rathke. Contextual equivalence for higher-order pi-calculus revisited. *Logical Methods in Computer Science*, 1(1), 2005.

[102] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, pages 1–53, mar 2014.

[103] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, Berlin, Germany, 2012.

[104] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, November 2006.

[105] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. A formal model of object mobility in resource-restricted deployment scenarios. In Farhad Arbab and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software*, volume 7253 of *Lecture Notes in Computer Science*, pages 187–204. Springer, Berlin, Germany, 2012.

[106] Kristján Valur Jónsson, Karl Palmskog, and Ymir Vigfússon. Secure distributed top-k aggregation. In *Communications (ICC), 2012 IEEE International Conference on*, pages 804–809. IEEE, June 2012.

[107] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[108] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: A comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.

[109] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220. ACM, 2009.

[110] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[111] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[112] KryoNet project website. URL `http://code.google.com/p/kryonet/`.

[113] Derek Kulinski, Jeff Burke, and Lixia Zhang. Video streaming over named data networking. In *IEEE COMSOC MMTC E-Letter*. IEEE, 2013.

[114] Zhifeng Lai, Shing-Chi Cheung, and Wing Kwong Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 235–244. ACM, 2010.

[115] Leslie Lamport. Distribution, 1987. URL `https://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt`. Retrieved 2014-08-13.

[116] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3): 382–401, July 1982.

[117] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed storage system. Technical report, Xerox Parc Research Center, Palo Alto, CA, 1976.

[118] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. A brief history of the internet. *ACM SIGCOMM Computer Communication Review*, 39(5):22–31, October 2009.

[119] Pierre Letouzey. Extraction in Coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, Berlin, Germany, 2008.

[120] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[121] Michael Lienhardt, Alan Schmitt, and Jean-Bernard Stefani. Oz/K: a kernel language for component-based open programming. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 43–52. ACM, 2007.

[122] Koon-Seng Lim, Constantin Adam, and Rolf Stadler. Decentralizing network management. Technical report, KTH Royal Institute of Technology, 2005.

[123] Koon-Seng Lim and Rolf Stadler. A navigation pattern for scalable internet management. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 405–420. IEEE, 2001.

[124] Barbara Liskov and Liuba Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 260–267. ACM, 1988.

[125] Peng Liu, Julian Dolby, and Charles Zhang. Finding incorrect compositions of atomicity. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 158–168, New York, NY, USA, 2013. ACM.

[126] Andreas Lochbihler. Type safe nondeterminism - a formal semantics of java threads. In *International Workshop on Foundations of Object-Oriented Languages*, FOOL 2008, January 2008.

[127] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically inferring multivariable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 103–116. ACM, 2007.

[128] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339. ACM, 2008.

[129] Shan Lu, Soyeon Park, and Yuanyuan Zhou. Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Transactions on Parallel and Distributed Systems*, 23(6):1060–1072, 2012.

[130] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, 27(1): 26–35, 2007.

[131] Lucene website. URL `http://lucene.apache.org`. Retrieved 2014-08-13.

[132] Andreas Lundblad. *Inlined Reference Monitors: Certification,Concurrency and Tree Based Monitoring*. PhD thesis, KTH Royal Institute of Technology, 2013.

[133] Claude Marche, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *The Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004. Formal Methods for Smart Cards.

[134] Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek. Detecting deadlock in programs with data-centric synchronization. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 322–331. IEEE, May 2013.

[135] Ilias Marinos, Robert N. M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 9:1–9:7. ACM, 2013.

[136] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[137] The Mathematical Components project website. URL `http://www.msr-inria.fr/projects/mathematical-components-2/`. Retrieved 2014-08-18.

[138] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[139] Kirill Mechitov, Reza Razavi, and Gul Agha. Architecture design principles to support adaptive service orchestration in WSN applications. *ACM SIGBED Review*, 4(3):37–42, July 2007.

[140] Elinor Mills. Web traffic redirected to china still a mystery, October 2010. URL `http://www.cnet.com/news/web-traffic-redirected-to-china-still-a-mystery/`. Retrieved 2014-08-25.

[141] Robin Milner. *Communication and concurrency*. Prentice Hall, Hertfordshire, United Kingdom, 1989.

[142] Robin Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89, 1993.

[143] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, New York, NY, USA, May 1999.

[144] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[145] Ugo Montanari and Matteo Sammartino. Network conscious pi-calculus: A concurrent semantics. *Electronic Notes in Theoretical Computer Science*, 286 (0):291–306, 2012. Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII).

[146] Daekyeong Moon, Jad Naous, Junda Liu, Kyriakos Zarifis, Martin Casado, Teemu Koponen, Scott Shenker, and Lee Breslau. Bridging the software/hardware forwarding divide. Technical report, University of California at Berkeley, 2010. URL `http://nsl.cs.usc.edu/~kyriakos/pubs/flowcaching.pdf`.

[147] Gordon Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[148] Keiko Nakata and Andri Saar. Compiling cooperative task management to continuations. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, Lecture Notes in Computer Science, pages 95–110. Springer, Berlin, Germany, 2013.

[149] Named Data Networking project website. URL `http://named-data.net`. Retrieved 2014-09-01.

[150] Uwe Nestmann, Rachele Fuzzati, and Massimo Merro. Modeling consensus in a process calculus. In Roberto Amadio and Denis Lugiez, editors, *CONCUR 2003 - Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 399–414. Springer, Berlin, Germany, 2003.

[151] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. Basic observables for a calculus for global computing. *Information and Computation*, 205(10): 1491–1525, 2007.

[152] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. Global computing in a dynamic network of tuple spaces. *Science of Computer Programming*, 64(2):187–204, 2007. Special Issue on Coordination Models and Languages (COORDINATION 2005).

[153] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.

[154] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *Communications Surveys Tutorials, IEEE*, 16(3):1617–1634, March 2014.

[155] OTP Design Principles User's Guide. URL `http://www.erlang.org/doc/design_principles/des_princ.html`. Retrieved 2014-08-25.

[156] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[157] Karl Palmskog. Shutdown protocol Promela model, Ott definition, and Coq proof scripts. URL `http://www.csc.kth.se/~palmskog/shutdown/`.

[158] Karl Palmskog, Mads Dam, Andreas Lundblad, and Ali Jafari. ABS-NET programs. URL `http://www.csc.kth.se/~palmskog/abs-net/`.

[159] Karl Palmskog, Mads Dam, Andreas Lundblad, and Ali Jafari. ABS-NET: Fully decentralized runtime adaptation for distributed objects. In Marco Carbone, Ivan Lanese, Alberto Lluch Lafuente, and Ana Sokolova, editors, *Proceedings 6th Interaction and Concurrency Experience*, volume 131 of *Electronic Proceedings in Theoretical Computer Science*, pages 85–100. Open Publishing Association, 2013.

[160] Karl Palmskog, Alberto Gonzalez Prieto, Catalin Meirosu, Rolf Stadler, and Mads Dam. Scalable metadata-directed search in a network of information. In *Future Network and Mobile Summit*, pages 1–8. IEEE, June 2010.

[161] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[162] Joachim Parrow and Peter Sjödin. Designing a multiway synchronization protocol. *Computer Communications*, 19(14):1151–1160, 1996.

[163] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[164] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice.* Addison-Wesley Professional, Boston, MA, USA, 2005.

[165] James Pelkey. A history of computer communications 1968-1988, chapter 2, 2007. URL `http://www.historyofcomputercommunications.info/Book/2/2.1-IntergalacticNetwork_1962-1964.html`. Retrieved 2014-07-15.

[166] C. Perkins. IP Mobility Support. RFC 2002 (Proposed Standard), October 1996. URL `http://www.ietf.org/rfc/rfc2002.txt`. Obsoleted by RFC 3220, updated by RFC 2290.

[167] C. Perkins. IP Mobility Support for IPv4, Revised. RFC 5944 (Proposed Standard), November 2010. URL `http://www.ietf.org/rfc/rfc5944.txt`.

[168] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, Cambridge, MA, USA, 2000.

[169] Andrew M. Pitts. Howe's method for higher-order languages. In Davide Sangiorgi and Jan J. M. M. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge Tracts in Theoretical Computer Science*, chapter 5, pages 197–232. Cambridge University Press, Cambridge, United Kingdom, October 2011.

[170] Ingmar Poese, Steve Uhlig, Mohamed Ali Kaafar, Benoit Donnet, and Bamba Gueye. IP geolocation databases: Unreliable? *ACM SIGCOMM Computer Communication Review*, 41(2):53–56, April 2011.

[171] Karl R. Popper. *The Poverty of Historicism.* The Beacon Press, Boston, MA, USA, 1957.

[172] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium.* USENIX Association, August 2003.

[173] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[174] Nadia Ranaldo and Eugenio Zimeo. Analysis of different future objects update strategies in proactive. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '07, pages 1–7. IEEE, march 2007.

[175] RAND Corporation. Paul Baran and the origins of the Internet. URL `http://www.rand.org/about/history/baran.list.html`. Retrieved 2014-07-15.

[176] Ruby on Rails website. URL `http://rubyonrails.org`. Retrieved 2014-08-13.

[177] Mark D. Ryan and Ben Smyth. Applied pi calculus. In Véronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, chapter 6. IOS Press, 2011.

[178] Andri Saar and Keiko Nakata. Compiling cooperative multitasking of abs to scala. Technical report, HATS project, 2012. URL `http://www.hats-project.eu/sites/default/files/D1.4/Paper1.pdf`.

[179] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, New York, NY, USA, 2011.

[180] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 33(1):5:1–5:69, January 2011.

[181] Davide Sangiorgi and David Walker. *Pi-Calculus: A Theory of Mobile Processes.* Cambridge University Press, New York, NY, USA, 2001.

[182] Ina Schaefer and Reiner Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.

[183] Jan Schäfer. *A Programming Model and Language for Concurrent and Distributed Object-Oriented Systems.* PhD thesis, University of Kaiserslautern, 2010.

[184] Peter Sewell, Paweł T. Wojciechowski, and Asis Unyapoth. Nomadic Pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Transactions on Programming Languages and Systems*, 32(4):12:1–12:63, April 2010.

[185] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20:71–122, jan 2010.

[186] Ankit Singla, P. Brighten Godfrey, Kevin Fall, Gianluca Iannaccone, and Sylvia Ratnasamy. Scalable routing on flat names. In *Proceedings of the 6th international conference on emerging networking experiments and technologies*, CoNEXT '10, pages 20:1–20:12. ACM, 2010.

[187] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9(3):219–228, May 1983.

[188] Diana Smetters and Van Jacobson. Securing network content. Technical report, PARC, October 2009.

[189] Gert Smolka. The definition of Kernel Oz. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer, Berlin, Germany, 1995.

[190] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[191] William N. Sumner, Christian Hammer, and Julian Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 161–176. Springer, Berlin, Germany, 2012.

[192] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. A unified semantics for future Erlang. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pages 23–32. ACM, 2010.

[193] Andrew Tanenbaum. *Computer Networks.* Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, USA, 4th edition, 2002.

[194] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[195] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do Scala developers mix the actor model with other concurrency models? In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 302–326. Springer, 2013.

[196] The Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013. URL `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`.

[197] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *2*, 1936.

[198] Maarten van Steen, Franz J. Hauck, Gerco Ballintijn, and Andrew S. Tanenbaum. Algorithmic design of the globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1998.

[199] Sebastian Vastag. Modeling quantitative requirements in SLAs with Network Calculus. In *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '11, pages 391–398, Brussels, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[200] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 334–345. ACM, 2006.

[201] Wei-Jen Wang and Carlos A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In Yeh-Ching Chung and José E. Moreira, editors, *Advances in Grid and Pervasive Computing*, volume 3947 of *Lecture Notes in Computer Science*, pages 360–372. Springer, Berlin, Germany, 2006.

[202] Shaun Waterman. Internet traffic was routed via Chinese servers, November 2010. URL `http://www.washingtontimes.com/news/2010/nov/15/internet-traffic-was-routed-via-chinese-servers/`. Retrieved 2014-08-25.

[203] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jaganathan. Accentuating the positive: Atomicity inference and enforcement using correct executions. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 19–34. ACM, 2011.

[204] Yi Wei and M. Brian Blake. Service-oriented computing and cloud computing: Challenges and opportunities. *IEEE Internet Computing*, 14(6):72–75, Nov 2010.

[205] Xalan website. URL `https://xml.apache.org/xalan-j/`. Retrieved 2014-08-13.

[206] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292–2330, 2008.

[207] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming ABCL/1. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 258–268. ACM, 1986.

[208] Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe. Type-safe runtime class upgrades in creol. In Roberto Gorrieri and Heike Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 202–217. Springer, Berlin, Germany, 2006.

[209] Haowei Yuan, Tian Song, and Patrick Crowley. Scalable NDN Forwarding: Concepts, Issues and Principles. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1–9. IEEE, 2012.

[210] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, Apr 1980.

Appendix A

Core ABS Syntax and Semantics

This appendix defines the syntax and semantics of Core ABS without cogs.

A.1 Syntax

The syntax consists of a functional level with algebraic data types and (side-effect free) functions, and an object level with interfaces, objects, methods and statements.

A.1.1 Functional Level

The definition of the syntax of the functional level of Core ABS is given in Table A.1; square brackets [ ] are used for optional elements. The use of an overline on a syntactic variable signifies a list of syntactic entities, as in $\overline{e}$ and $\overline{x}$. The delimiter for a list is implicit, but in most cases a comma. For variable and method type declarations, there is a slight abuse of notation for conciseness, namely, $\overline{T\ x\,;}$ is a possibly empty list $T_1\, x_1;\ ..\,;\ T_n x_n\,;$.

Syntactic categories			*Definitions*
T in Ground Type	T	$::=$	$B \mid I \mid D[\langle\overline{T}\rangle] \mid \mathbf{Fut}\langle T\rangle$
B in Basic Type	B	$::=$	$\mathsf{Bool} \mid \mathsf{Int} \mid \cdots$
A in Type	A	$::=$	$N \mid T \mid D\langle\overline{A}\rangle \mid \mathbf{Fut}\langle A\rangle$
N, I in Name	Dd	$::=$	$\mathbf{data}\ D[\langle\overline{N}\rangle] = Cons[\mid\overline{Cons}]\,;$
x in Variable	$Cons$	$::=$	$Co[(\overline{A})]$
e in Expression	F	$::=$	$\mathbf{def}\ A\ fn[\langle\overline{N}\rangle](\overline{A\ x}) = e\ ;$
b in Bool Expression	e	$::=$	$b \mid x \mid t \mid \mathbf{this} \mid \mathbf{destiny} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \mathbf{case}\ e\ \{\overline{br}\}$
t in Ground Term	$t,\ v$	$::=$	$Co[(\overline{t})] \mid \mathbf{null} \mid o \mid f \mid \mathbf{True} \mid \mathbf{False}$
br in Branch	br	$::=$	$p \Rightarrow e\,;$
p in Pattern	p	$::=$	$\_ \mid x \mid t \mid Co[(\overline{p})]$

Table A.1: Core ABS functional level syntax

Ground types include basic types such as Bool and Int, which can be considered built in, and also user-defined algebraic data types D and user-declared interfaces I. In contrast to ground types, a type A can contain type variables N, enabling polymorphism for data types and functions. In a data type declaration Dd, possibly parameterized with the type variables $\overline{N}$, there must be at least one constructor $Cons$, possibly with a list of parameters $\overline{A}$. Function declarations F, which again may be parameterized with variables $\overline{N}$, include a return type A, a list of function parameters $\overline{T\ x}$ with their types, and an expression e.

Expressions e are boolean expressions b, variables x, ground terms t (at the object level, referred to as values v), special variables **this** and **destiny**, data type constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$ and case branches **case** $e\ \{\overline{br}\}$. Boolean expressions are mentioned explicitly because of their use in the object level and its semantics, but do not differ significantly from expressions typed by user-defined data types. Such expressions can consist of variables, function expressions, and the ground terms **True** and **False**, composed by standard operators such as conjunction and disjunction. Patterns p can be used to decompose a constructor in a case branch, checking for term equality or binding variables to subterms. Exhaustiveness of case branches for a given case expression type is not enforced.

The functional level is intermingled with the object level in that interface names I are ground types, there are special expression keywords **this** and **destiny**, and there are special ground terms **null**, o and f. **null** plays the part of default value for interface and future types. A future value f has type $\mathbf{Fut}\langle T\rangle$, for some ground type T, capturing the fact that it is a placeholder for a yet-to-be-seen value of the type T. An object identifier o is generated at runtime during object instantiation and is typed by its class C.

A.1.2 Object Level

The object level syntax, shown in Table A.2, defines interfaces, classes, methods, object creation and method calls. A Core ABS program P defines data types, functions, interfaces, classes and a list of statements (main block) that is executed initially. An interface declaration IF consists of an interface name I and $\overline{Sg\,;}$, which, again by slight abuse of notation, is a possibly empty list of method signatures $Sg_1\,;\,..\,;Sg_n\,;$. A class declaration CL consists of a class name C, an optional list of interfaces $\overline{I}$ whose methods the class implements, and a list of method declarations $\overline{M}$. The first, optional, comma-separated list of variable-type declarations $\overline{T\ x}$ defines mandatory constructor parameters that must be given when instantiating an object of the class with **new** $C(\overline{e})$. The second, semicolon-separated such list declares the class fields, which assume the default values for their types on instantiation. Classes can optionally define a method named **init** to manually initialize the fields to other values.

A method signature Sg consists of a return type T, a method name m, and a list of parameter variables and their types. Methods M have a signature, a list

Syntactic categories			*Definitions*
C, m in Name	P	$::=$	$\overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\overline{T\ x}\,;\ \overline{s}\}$
g in Guard	IF	$::=$	$\textbf{interface}\ I\ \{\overline{Sg}\,;\}$
s in Statement	CL	$::=$	$\textbf{class}\ C\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{I}]\ \{\overline{T\ x}\,;\ \overline{M}\}$
	Sg	$::=$	$T\ m\,(\overline{T\ x})$
	M	$::=$	$Sg\ \{\overline{T\ x}\,;\ \overline{s}\ \textbf{return}\ e\,;\}$
	g	$::=$	$b \mid e? \mid g \wedge g$
	s	$::=$	$x = rhs; \mid \textbf{suspend}\,; \mid \textbf{await}\ g; \mid \textbf{skip}\,;$
			$\mid \textbf{if}\ b\ \{\overline{s}\}\ [\textbf{else}\ \{\overline{s}\}] \mid \textbf{while}\ b\ \{\overline{s}\}$
	rhs	$::=$	$e \mid \textbf{new}\ C(\overline{e}) \mid e!m(\overline{e}) \mid e.m(\overline{e}) \mid e.\,\textbf{get}$

Table A.2: Core ABS object level syntax

of declarations of local variables, a list of statements $\overline{s}$, and a single, final return statement.

An asynchronous method invocation statement $x = e!m(\overline{e})\,;$ does not block, and assigns a future identifier to the variable x, with e reducing to the callee's identifier and $\overline{e}$ the argument list. The actual return value of the invocation can later be retrieved into the variable y by the assignment $y = x.\,\textbf{get}\,;$ which possibly blocks. Synchronous method invocation $x = e.m(\overline{e})\,;$ can block and assigns a value of the method's return type directly. The special **this** variable can be used by a class to call internal methods, which are possibly not defined in any interface.

Guards g consist of ordinary boolean conditions b and special tests $e?$, which check that the future value e reduces to is resolved, i.e., that the associated method invocation has finished. If a tested future is unresolved, the guard is false and the current process is suspended, making it possible for other processes to execute. The statement **suspend**; allows direct, unconditional suspension.

A.2 Type System of Core ABS

The type system of Core ABS can be divided into one part for the functional level and one part for the object level, with the latter building heavily upon the former.

A.2.1 Functional Level

The functional level well-typing relation, defined in Figure A.1, is given with respect to a typing context in the form of a finite map Γ from variables and constants to types. A lookup on the variable x in Γ is given by $\Gamma(x)$. The addition to Γ of a binding of x to T is given by $\Gamma[x \mapsto T]$; an existing binding for x to another type in Γ is not relevant in the resulting map. Two finite maps Γ and Γ' can be composed to form a map $\Gamma \circ \Gamma'$, so that $\Gamma \circ \Gamma'(x) = \Gamma'(x)$ whenever there is binding for x in Γ', and $\Gamma \circ \Gamma'(x) = \Gamma(x)$ otherwise. Γ and Γ' are in the extension relation, $\Gamma \subseteq \Gamma'$, whenever Γ' has bindings for all keys with bindings in Γ, and the corresponding types coincide. $[\,]$ is the empty map, while $[x \mapsto T]$ is the map with the single binding of x to T.

$$\text{(T-ConsDecl)}\ \frac{\Gamma(Co) = \overline{A} \rightarrow D[\langle\overline{N}\rangle]}{\Gamma \vdash Co(\overline{A}) : D[\langle\overline{N}\rangle]} \qquad \text{(T-DataDecl)}\ \frac{\Gamma \vdash \overline{Cons} : D[\langle\overline{N}\rangle]}{\Gamma \vdash \mathbf{data}\ D[\langle\overline{N}\rangle] = \overline{Cons}\,;} \qquad \text{(T-Sub)}\ \frac{\Gamma \vdash e : T \quad T \preceq T'}{\Gamma \vdash e : T'}$$

$$\text{(T-Case)}\ \frac{\begin{array}{c}\Gamma \vdash e : A\\ \Gamma \vdash \overline{br} : A \rightarrow A'\end{array}}{\Gamma \vdash \mathbf{case}\ e\ \{\overline{br}\} : A'} \qquad \text{(T-FuncExpr)}\ \frac{\begin{array}{c}\mathtt{tmatch}\,(\overline{A}, \overline{A}') = \sigma \quad \sigma \neq \bot\\ \Gamma \vdash \overline{e} : \overline{A}' \quad \Gamma(fn) = \overline{A} \rightarrow A\end{array}}{\Gamma \vdash fn(\overline{e}) : A\,\sigma} \qquad \text{(T-ConsExpr)}\ \frac{\begin{array}{c}\Gamma \vdash \overline{e} : \overline{A}' \quad \sigma \neq \bot\\ \mathtt{tmatch}\,(\overline{A}, \overline{A}') = \sigma\\ \Gamma(Co) = \overline{A} \rightarrow D[\langle\overline{N}\rangle]\end{array}}{\Gamma \vdash Co(\overline{e}) : D[\langle\overline{N}\rangle]\,\sigma}$$

$$\text{(T-ObjectId)}\ \frac{\Gamma(o) = C}{\Gamma \vdash o : C} \qquad \text{(T-FutureId)}\ \frac{\Gamma(f) = \mathbf{Fut}\,\langle T\rangle}{\Gamma \vdash f : \mathbf{Fut}\,\langle T\rangle} \qquad \text{(T-Wildcard)}\ \frac{}{\Gamma \vdash \_ : A} \qquad \text{(T-Bool)}\ \frac{}{\Gamma \vdash b : \mathsf{Bool}} \qquad \text{(T-Null)}\ \frac{}{\Gamma \vdash \mathbf{null} : A}$$

$$\text{(T-FuncDecl)}\ \frac{\begin{array}{c}\Gamma(fn) = A_1, .., A_n \rightarrow A\\ \Gamma[x_1 \mapsto A_1, .., x_n \mapsto A_n] \vdash e : A\end{array}}{\Gamma \vdash \mathbf{def}\ A\ fn[\langle\overline{N}\rangle](A_1\,x_1, .., A_n\,x_n) = e\,;} \qquad \text{(T-Branch)}\ \frac{\begin{array}{c}\Gamma' \vdash p : A \quad \Gamma' \vdash e : A'\\ \Gamma' = \Gamma \circ \mathtt{psubst}\,(p, A, \Gamma)\end{array}}{\Gamma \vdash p \Rightarrow e\,;\ :\ A \rightarrow A'} \qquad \text{(T-Var)}\ \frac{\Gamma(x) = A}{\Gamma \vdash x : A}$$

Figure A.1: Core ABS functional level type system

An initial typing context is assumed to map the names of the data types and function declarations under consideration to appropriate types, which is reflected in the rules T-ConsDecl and T-FuncDecl. The rule T-Null allows the **null** term to have any type. The rule T-Var types a variable according to the type recorded for it in the context, as is done in, e.g., T-FuncDecl. The `tmatch` auxiliary function, used in T-FuncExpr and T-ConsExpr, attempts to match the type variables of the formal parameter types to the actual parameter types; if there is no match, `tmatch` returns $\bot$. The `psubst` auxiliary function, used in the rule T-Branch, constructs a typing context for which a pattern must be well-typed. If A is a type variable N, then p is a variable x and $\mathtt{psubst}(p, N, \Gamma) = [x \rightarrow N]$. If, on the other hand, A is not a type variable, we define the result based on the structure of p. If $p = x$ and $\Gamma(x) = T$, then $\mathtt{psubst}(p, N, \Gamma) = [\,]$. If $p = x$ and Γ has no binding for x, then $\mathtt{psubst}(p, N, \Gamma) = [p \mapsto A]$. If $p = t$ or $p = \_$, then $\mathtt{psubst}(p, N, \Gamma) = [\,]$. Finally, if $p = Co(p_1, .., p_n)$ and $\Gamma(Co) = A_1, .., A_n \rightarrow A$, then $\mathtt{psubst}(p, N, \Gamma) = \mathtt{psubst}(p_1, A_1, \Gamma) \circ .. \circ \mathtt{psubst}(p_n, A_n, \Gamma)$. The subtyping relation $\preceq$, used in the rule T-Sub, is such that $C \preceq I$ whenever the class C implements the interface I. The relation also anticipates extension to interface subtyping via inheritance, which is present in full ABS but not Core ABS.

A.2.2 Object Level

The well-typing relation of the object level, shown in Figure A.2, proceeds in a straightforward way on the syntactic structure of the programs, for the most part.

In the rule T-Program, a program is well-typed with respect to a context when its all data declarations, function declarations and class declarations are well-typed; typing of interfaces is simpler and is omitted. For the typing of the statement list $\overline{s}$, all variable declarations are added to the context.

$$\text{(T-Poll)}\ \frac{\Gamma \vdash e : \textbf{Fut}\,\langle T\rangle}{\Gamma \vdash e? : \mathsf{Bool}} \qquad \text{(T-Get)}\ \frac{\Gamma \vdash e : \textbf{Fut}\,\langle T\rangle}{\Gamma \vdash e.\textbf{get} : T} \qquad \text{(T-Skip)}\ \frac{}{\Gamma \vdash \textbf{skip}\,;} \qquad \text{(T-Await)}\ \frac{\Gamma \vdash g : \mathsf{Bool}}{\Gamma \vdash \textbf{await}\, g;} \qquad \text{(T-Suspend)}\ \frac{}{\Gamma \vdash \textbf{suspend}\,;}$$

$$\text{(T-Assign)}\ \frac{\Gamma \vdash x : T \quad \Gamma \vdash \mathit{rhs} : T}{\Gamma \vdash x = \mathit{rhs};} \qquad \text{(T-And)}\ \frac{\Gamma \vdash g : \mathsf{Bool} \quad \Gamma \vdash g' : \mathsf{Bool}}{\Gamma \vdash g \wedge g' : \mathsf{Bool}} \qquad \text{(T-New)}\ \frac{\Gamma \vdash \overline{e} : \mathtt{ptypes}\,(C) \quad I \in \mathtt{interfaces}\,(C)}{\Gamma \vdash \textbf{new}\, C(\overline{e}) : I} \qquad \text{(T-AsyncCall)}\ \frac{\Gamma \vdash e.m(\overline{e}) : T}{\Gamma \vdash e!m(\overline{e}) : \textbf{Fut}\,\langle T\rangle}$$

$$\text{(T-Conditional)}\ \frac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash \overline{s} \quad [\Gamma \vdash \overline{s'}]}{\Gamma \vdash \textbf{if}\ b\ \{\overline{s}\}\ [\textbf{else}\ \{\overline{s'}\}]} \qquad \text{(T-While)}\ \frac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash \overline{s}}{\Gamma \vdash \textbf{while}\ b\ \{\overline{s}\}} \qquad \text{(T-SyncCall)}\ \frac{\Gamma \vdash e : I \quad \Gamma \vdash \overline{e} : \overline{T} \quad \mathtt{match}\,(m, \overline{T} \to T, I)}{\Gamma \vdash e.m(\overline{e}) : T}$$

$$\text{(T-Method)}\ \frac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto T_1, .., x_i \mapsto T_i, x'_1 \mapsto T'_1, .., x'_j \mapsto T'_j] \\ \Gamma'' = \Gamma'[\textbf{destiny} \mapsto \textbf{Fut}\,\langle T\rangle] \quad \Gamma'' \vdash \overline{s} \quad \Gamma'' \vdash e : T\end{array}}{\Gamma \vdash T\, m\,(T_1\, x_1, .., T_i\, x_i)\ \{T'_1\, x'_1; ..; T'_j\, x'_j\,; \overline{s}\ \textbf{return}\ e\,;\}}$$

$$\text{(T-Class)}\ \frac{[\forall I \in \overline{I}.\ \mathtt{implements}\,(C, I)] \quad \Gamma[\textbf{this} \mapsto C,\ \mathtt{fields}\,(C)] \vdash \overline{M}}{\Gamma \vdash \textbf{class}\ C\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{I}]\ \{\overline{T'\ x'}\ ;\ \overline{M}\}}$$

$$\text{(T-Program)}\ \frac{\Gamma[x_1 \mapsto T_1, .., x_n \mapsto T_n] \vdash \overline{s} \quad \forall Dd \in \overline{Dd}.\ \Gamma \vdash Dd \quad \forall F \in \overline{F}.\ \Gamma \vdash F \quad \forall CL \in \overline{CL}.\ \Gamma \vdash CL}{\Gamma \vdash \overline{Dd}\,\overline{F}\,\overline{IF}\,\overline{CL}\ \{T_1\, x_1; ..; T_n\, x_n\ ;\ \overline{s}\}}$$

Figure A.2: Core ABS object level type system

In the rule T-Method, parameter declarations and local variable declarations are again added the context before deferring to the well-typedness of the statement list and return expression. In addition, a binding is added for the special expression **destiny** to the type for the return expression e, so that **destiny** can be used as a variable containing the identifier of the future a method invocation produces.

The rule T-Poll formalizes the requirement that a future resolution test must be performed on an expression that actually reduces to a future identifier, as does T-Get for the **.get** operator. In T-New, for a given class identifier C, the auxiliary function `ptypes` returns the class parameter types and `interfaces` returns the set of identifiers of the interfaces which the class implements. The effect of the rule is that all variables containing object identifiers must be typed with an interface which the class of the object implements—not the implementing class itself.

In T-Class, the auxiliary function `implements` is used to check that the given class properly implements the methods of all the interfaces in the list $\overline{I}$. The auxiliary function `fields` produces type bindings for parameters and fields for a given class. The addition of a binding for the special expression **this** allows methods to call class-internal methods, by masquerading the class name C as an interface type. In T-SyncCall, the auxiliary function `match` checks that types of the given expression list of arguments coincides with the actual parameter types specified for the method in the interface or class. T-AsyncCall expresses the typing of asynchronous method invocations as having the future type of the corresponding synchronous invocation. The rules for skip statements, suspend statements, conditionals, and while loops are standard.

A.3 Operational Semantics

In the standard operational semantics of Core ABS, a runtime configuration consists of objects, futures and method invocations. In this section, a reduction system for functional (side-effect free) expressions paves the way to a transition system that describes how the entities evolve and interact. The transition system rules apply to subsets of a global configuration, modulo rearrangement of entities to fit the left-hand side of the rules, as in the Maude style of modeling distributed systems [40]. The execution of a program is a possibly infinite sequence of global configurations, such that the transition between a previous configuration and the next is valid in the system.

A.3.1 Runtime Configurations

In the standard runtime syntax, shown in Table A.3, configurations *cn*, consisting of futures, objects, and method invocations (*fut*, *object*, *invoc*), are composed via a whitespace operator, with ϵ being the empty configuration. A global configuration is shown inside curly brackets, e.g., $\{cn\}$.

cn	$::=$	$\epsilon \mid fut \mid object \mid invoc \mid cn\ cn$	pr	$::=$	$process \mid \textbf{idle}$
fut	$::=$	$\mathsf{fut}\,(f, val)$	a, l	$::=$	$\epsilon \mid T\ x\ v \mid a, a$
$object$	$::=$	$\mathsf{ob}\,(o, a, pr, q)$	val	$::=$	$v \mid \bot$
$process$	$::=$	$\{l \mid \overline{sp}\} \mid \textbf{error}$	sp	$::=$	$\textbf{return}\ e\,; \mid \textbf{cont}\,(f)\,; \mid s$
$invoc$	$::=$	$\mathsf{invoc}\,(o, f, m, \overline{v})$	q	$::=$	$\emptyset \mid process \mid q\ q$

Table A.3: Core ABS standard runtime syntax

A future $\mathsf{fut}\,(f, val)$ has a future identifier f and, by the definition of val, either a resolved value (ground term) v, or $\bot$ to indicate its status of being unresolved. An object $\mathsf{ob}\,(o, a, pr, q)$ has an object identifier o, a store a for its field types and values, an active process pr and a pool q of suspended processes. A normal process *process* has a store l of local variable types and values, and a list of process statements. ϵ refers also to the empty store, disambiguated from the empty configuration by context. Process statements sp are statements s extended with a return statement **return** e ; and a continuation statement **cont** (f) ;. A method invocation $\mathsf{invoc}\,(o, f, m, \overline{v})$ contains the intended recipient object's identifier o, the associated future identifier f, the method name m, and a list of argument values $\overline{v}$. Data types, functions, interfaces and classes are not represented explicitly in runtime configurations, since they are assumed to be static throughout execution.

A.3.2 Reduction System for Functional Expressions

Given a substitution σ binding variables to terms, functional expressions can be reduced to terms in accordance with the reduction system shown in Figure A.3, which defines the reduction relation $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$. The relation intuitively holds when the expression e in the context σ can be reduced to the expression e' in the context σ'. There is no guarantee that a sequence of valid reductions eventually leads to a ground term; recursive functions can lead to infinite reduction sequences, and incomplete case branch coverage can make the sequence halt on an expression that is not a ground term.

$$\frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \le i \le n}{\sigma \vdash Co(e_1, .., e_i, .., e_n) \rightsquigarrow \sigma' \vdash Co(e_1, .., e_i', .., e_n)}\ (\text{RedCons}) \qquad \frac{\sigma(x) = t}{\sigma \vdash x \rightsquigarrow \sigma \vdash t}\ (\text{RedVar})$$

$$\frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \le i \le n}{\sigma \vdash fn(e_1, .., e_i, .., e_n) \rightsquigarrow \sigma' \vdash fn(e_1, .., e_i', .., e_n)}\ (\text{RedFunExp})$$

$$\frac{\sigma \vdash e \rightsquigarrow \sigma' \vdash e'}{\sigma \vdash \textbf{case}\, e\,\{\overline{br}\} \rightsquigarrow \sigma' \vdash \textbf{case}\, e'\,\{\overline{br}\}}\ (\text{RedCase1}) \qquad \frac{\texttt{match}\,(\sigma(p), t) = \bot}{\sigma \vdash \textbf{case}\, t\,\{p \Rightarrow e\,;\, \overline{br}\} \rightsquigarrow \sigma \vdash \textbf{case}\, t\,\{\overline{br}\}}\ (\text{RedCase3})$$

$$\frac{\begin{array}{l}\texttt{match}\,(\sigma(p), t) = \sigma' \quad \sigma' \neq \bot \quad \texttt{vars}\,(\sigma(p)) = \{x_1, \dots, x_n\} \\ \texttt{fresh}\,(\{y_1, \dots, y_n\}) \quad \sigma'' = \sigma[y_1 \mapsto \sigma'(x_1), .., y_n \mapsto \sigma'(x_n)]\end{array}}{\sigma \vdash \textbf{case}\, t\,\{p \Rightarrow e;\, \overline{br}\} \rightsquigarrow \sigma'' \vdash e[x_1 \mapsto y_1, .., x_n \mapsto y_n]}\ (\text{RedCase2})$$

$$\frac{\overline{x}_{fn} = x_1, \dots, x_n \quad \texttt{fresh}\,(\{y_1, \dots, y_n\})}{\sigma \vdash fn(t_1, \dots, t_n) \rightsquigarrow \sigma[y_1 \mapsto t_1, .., y_n \mapsto t_n] \vdash e_{fn}[x_1 \mapsto y_1, .., x_n \mapsto y_n]}\ (\text{RedFunGround})$$

Figure A.3: Core ABS reduction rules for functional expressions

Evaluation of a function expression involving the function fn, as defined in the rules RedFunExp and RedFunGround, proceeds by first reducing all argument expressions to terms. Suppose the list of parameter variables in the declaration of fn is $x_1, \dots, x_n$ and the expression in the declaration is e_{fn}. The function expression is then replaced with the expression that results from syntactically replacing the parameters in e_{fn} with the new, unique names $y_1, \dots, y_n$, and the substitution is extended to include bindings from the new names to the respective terms.

Let $\texttt{dom}(\sigma)$ be the set of names which are bound in a substitution σ, and let `vars` be the function which returns the set of variables in a pattern. The `match` function in the rules RedCase2 and RunCase3, given a pattern p and a term t, returns a substitution σ such that $\sigma(p) = t$ and $\texttt{dom}(\sigma) = \texttt{vars}(p)$ if it exists, and $\perp$ otherwise. We define a substitution σ as well-typed for a given context Γ, written $\Gamma \vdash \sigma$, whenever, for all x bound in σ, $\Gamma \vdash \sigma(x) : \Gamma(x)$. This definition is used to state the type preservation property of Lemma A.3.1.

Lemma A.3.1 (Type Preservation). *Let Γ be a typing context and σ be a substitution such that $\Gamma \vdash \sigma$. If $\Gamma \vdash e : A$ and $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$, then there is a typing context Γ' such that $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash \sigma'$ and $\Gamma' \vdash e' : A$.*

Proof. By induction on rule applications [103]. □

A.3.3 Operational Semantics of Concurrent Objects

Guard expressions are not covered by the reduction system for functional expressions in the previous subsection. Guard evaluation is different since it potentially depends on the state of a global configuration. Figure A.4 shows the reduction rules for guards. Below, the boolean result of evaluating a guard expression g with respect to a configuration cn and a store a, if it exists, is written as $[\![g]\!]_a^{cn}$. Similarly, the ground term result of evaluating an expression e with respect to a store a, if it exists, is written as $[\![e]\!]_a$.

$$\text{(RedBoolGuard)}\quad \frac{\sigma \vdash b \rightsquigarrow \sigma \vdash b'}{\begin{array}{c}\sigma, cn \vdash b\\ \rightsquigarrow \sigma, cn \vdash b'\end{array}} \qquad \text{(RedReplyGuard1)}\quad \frac{\begin{array}{l}\sigma \vdash e \rightsquigarrow \sigma \vdash f\\ \mathsf{fut}(f, v) \in cn\end{array}}{\begin{array}{c}\sigma, cn \vdash e?\\ \rightsquigarrow \sigma, cn \vdash \mathbf{True}\end{array}} \qquad \text{(RedReplyGuard2)}\quad \frac{\begin{array}{l}\sigma \vdash e \rightsquigarrow \sigma \vdash f\\ \mathsf{fut}(f, \perp) \in cn\end{array}}{\begin{array}{c}\sigma, cn \vdash e?\\ \rightsquigarrow \sigma, cn \vdash \mathbf{False}\end{array}}$$

$$\text{(RedGuards)}\quad \frac{\sigma, cn \vdash g_1 \rightsquigarrow \sigma, cn \vdash g_1' \quad \sigma, cn \vdash g_2 \rightsquigarrow \sigma, cn \vdash g_2'}{\sigma, cn \vdash g_1 \wedge g_2 \rightsquigarrow \sigma, cn \vdash g_1' \wedge g_2'}$$

Figure A.4: Core ABS reduction rules for guard expressions

The set of transition rules for configurations is split between Figure A.5 and Figure A.6. The rule Suspend puts the current active process into the pool of inactive processes, allowing another process to run with the help of rule Activate. The `select` auxiliary function decides the process to make active for an idle object, given the complete global state. The function is implementation-specific and thus left unspecified, in effect providing a hook for different schedulers. $q \setminus \mathit{process}$ is the pool q with process *process* removed, and $q \cup \mathit{process}$ is q with *process* added. If a is a store, then $a[x \mapsto v]$ is the store that results when replacing the value for the variable x with v in a.

In Bind-Mtd, the `bind` auxiliary function produces a process from a method invocation, retrieving the statements to execute and initializing local variables in the process store, among them **destiny**, which is assigned the future identifier f.

The resulting process is then put in the pool, and the invocation removed. The rule Return subsequently uses the value stored in **destiny** to find the future to resolve in the configuration. The `class` auxiliary function, which takes an object identifier as an argument, returns the class associated with the identifier. Note that `bind` returns the process **error** if the class does not have the method indicated, or there is an argument-parameter mismatch.

$$\textsc{(Assign-Local)}\quad \frac{x \in \texttt{dom}\,(l) \quad [\![e]\!]_{a \circ l} = v}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid x = e;\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l[x \mapsto v] \mid \overline{sp}\}, q)\end{array}}$$

$$\textsc{(Assign-Field)}\quad \frac{x \in \texttt{dom}\,(a) \quad [\![e]\!]_{a \circ l} = v}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid x = e;\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a[x \mapsto v], \{l \mid \overline{sp}\}, q)\end{array}}$$

$$\textsc{(Idle)}\quad \frac{}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \}, q)\\ \to \mathsf{ob}\,(o, a, \textbf{idle}, q)\end{array}}$$

$$\textsc{(Cond-True)}\quad \frac{[\![b]\!]_{a \circ l} = \textbf{True}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \textbf{if}\ b\,\{\overline{s}\}\ [\textbf{else}\,\{\overline{s'}\}]\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l \mid \overline{s}\ \overline{sp}\}, q)\end{array}}$$

$$\textsc{(Cond-False)}\quad \frac{[\![b]\!]_{a \circ l} = \textbf{False}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \textbf{if}\ b\,\{\overline{s}\}\ [\textbf{else}\,\{\overline{s'}\}]\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l \mid [\overline{s'}]\ \overline{sp}\}, q)\end{array}}$$

$$\textsc{(Read-Fut)}\quad \frac{[\![e]\!]_{a \circ l} = f}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid x = e.\textbf{get};\ \overline{sp}\}, q)\ \mathsf{fut}\,(f, v)\\ \to \mathsf{ob}\,(o, a, \{l \mid x = v;\ \overline{sp}\}, q)\ \mathsf{fut}\,(f, v)\end{array}}$$

$$\textsc{(Bind-Mtd)}\quad \frac{\texttt{bind}\,(o, f, m, \overline{v},\ \texttt{class}\,(o)) = \mathit{process}}{\begin{array}{c}\mathsf{ob}\,(o, a, p, q)\ \mathsf{invoc}\,(o, f, m, \overline{v})\\ \to \mathsf{ob}\,(o, a, p, q \cup \mathit{process})\end{array}}$$

$$\textsc{(Await-True)}\quad \frac{[\![g]\!]^{cn}_{a \circ l} = \textbf{True}}{\begin{array}{c}\{\mathsf{ob}\,(o, a, \{l \mid \textbf{await}\,g;\ \overline{sp}\}, q)\ cn\}\\ \to \{\mathsf{ob}\,(o, a, \{l \mid \overline{sp}\}, q)\ cn\}\end{array}}$$

$$\textsc{(Await-False)}\quad \frac{[\![g]\!]^{cn}_{a \circ l} = \textbf{False}}{\begin{array}{c}\{\mathsf{ob}\,(o, a, \{l \mid \textbf{await}\,g;\ \overline{sp}\}, q)\ cn\}\\ \to \{\mathsf{ob}\,(o, a, \{l \mid \textbf{suspend}\,;\ \textbf{await}\,g;\ \overline{sp}\}, q)\ cn\}\end{array}}$$

$$\textsc{(Skip)}\quad \frac{}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \textbf{skip}\,;\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l \mid \overline{sp}\}, q)\end{array}}$$

$$\textsc{(Async-Call)}\quad \frac{[\![e]\!]_{a \circ l} = o' \quad [\![\overline{e}]\!]_{a \circ l} = \overline{v} \quad \texttt{fresh}\,(f)}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid x = e!m(\overline{e});\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l \mid x = f;\ \overline{sp}\}, q)\ \mathsf{invoc}\,(o', f, m, \overline{v})\ \mathsf{fut}\,(f, \bot)\end{array}}$$

$$\textsc{(Return)}\quad \frac{[\![e]\!]_{a \circ l} = v \quad l(\,\textbf{destiny}\,) = f}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \textbf{return}\,e;\ \overline{sp}\}, q)\ \mathsf{fut}\,(f, \bot)\\ \to \mathsf{ob}\,(o, a, \{l \mid \overline{sp}\}, q)\ \mathsf{fut}\,(f, v)\end{array}}$$

$$\textsc{(Suspend)}\quad \frac{}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \textbf{suspend}\,;\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \textbf{idle}, q \cup \{l \mid \overline{sp}\})\end{array}}$$

$$\textsc{(Activate)}\quad \frac{\texttt{select}\,(q, a, cn) = \mathit{process}}{\{\mathsf{ob}\,(o, a, \textbf{idle}, q)\ cn\} \to \{\mathsf{ob}\,(o, a, \mathit{process}, q \setminus \mathit{process})\ cn\}}$$

Figure A.5: Standard Core ABS reduction rules of concurrent objects, part 1.

In New-Object, `fresh`(o') ensures that the identifier o' is globally unique, `init` produces a process for the initializing method of the class (or an empty process if such a method does not exist), and `atts` sets the field values for the new object, including the special field **this** which is given the value o'. In Rem-Sync-Call, the new, fresh variable y is introduced to hold the future identifier associated with an asynchronous call; to type the variable properly at run time in the store, the method's return type is looked up in the class using the `returns` auxiliary function.

The rules Self-Sync-Call and Self-Sync-Return-Sched provide the justification for

$$\dfrac{\begin{array}{l}(\textsc{Self-Sync-Call})\\ l(\,\mathbf{destiny}\,) = f' \quad [\![e]\!]_{a \circ l} = o \quad [\![\overline{e}]\!]_{a \circ l} = \overline{v}\\ \texttt{fresh}\,(f) \quad \texttt{bind}\,(o, f, m, \overline{v}, \texttt{class}\,(o)) = \{l' \mid \overline{sp}'\}\end{array}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid x = e.m(\overline{e});\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l' \mid \overline{sp}'\,\mathbf{cont}\,(f'); \}, q \cup \{l \mid x = f.\,\mathbf{get};\ \overline{sp}\})\ \mathsf{fut}\,(f, \bot)\end{array}}$$

$$\dfrac{\begin{array}{l}(\textsc{Self-Sync-Return-Sched})\\ l'(\,\mathbf{destiny}\,) = f\end{array}}{\mathsf{ob}\,(o, a, \{l \mid \mathbf{cont}\,(f); \}, q \cup \{l' \mid \overline{sp}\}) \to \mathsf{ob}\,(o, a, \{l' \mid \overline{sp}\}, q)}$$

$$\dfrac{\begin{array}{l}(\textsc{New-Object})\\ \texttt{fresh}\,(o') \quad \texttt{init}\,(C) = \mathit{process} \quad \texttt{atts}\,(C, [\![\overline{e}]\!]_{a \circ l}, o') = a'\end{array}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l | x = \mathbf{new}\ C(\overline{e});\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l | x = o';\ \overline{sp}\}, q)\ \mathsf{ob}\,(o', a', \mathbf{idle}, \mathit{process})\end{array}}$$

$$\dfrac{\begin{array}{l}(\textsc{While-True})\\ [\![b]\!]_{a \circ l} = \mathbf{True}\end{array}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \mathbf{while}\ b\, \{\overline{s}\}\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l \mid \overline{s}\ \mathbf{while}\ b\, \{\overline{s}\}\ \overline{sp}\}, q)\end{array}} \qquad \dfrac{\begin{array}{l}(\textsc{While-False})\\ [\![b]\!]_{a \circ l} = \mathbf{False}\end{array}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \mathbf{while}\ b\, \{\overline{s}\}\ \overline{sp}\}, q)\\ \to \mathsf{ob}\,(o, a, \{l \mid \overline{sp}\}, q)\end{array}}$$

$$\dfrac{\begin{array}{l}(\textsc{Rem-Sync-Call})\\ [\![e]\!]_{a \circ l} = o' \quad \texttt{fresh}\,(y) \quad \texttt{returns}\,(\,\texttt{class}\,(o'), m) = T\end{array}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid x = e.m(\overline{e});\ \overline{sp}\}, q)\ \mathsf{ob}\,(o', a', pr, q')\\ \to \mathsf{ob}\,(o, a, \{l, \mathbf{Fut}\,\langle T\rangle\ y\ \mathbf{null} \mid y = o'!m(\overline{e});\ x = y.\,\mathbf{get};\ \overline{sp}\}, q)\ \mathsf{ob}\,(o', a', pr, q')\end{array}}$$

Figure A.6: Standard Core ABS reduction rules of concurrent objects, part 2.

extending process statements with a continuation statement. When an object calls itself in Self-Sync-Call, control passes to a new process and must then somehow be passed back. Therefore, a continuation statement containing the future identifier of the caller process is added to the callee process, and eventually consumed through the rule Self-Sync-Return-Sched.

The runtime typing rules in Figure A.7, distinguished by the turnstile subscript R and the suffix **ok** in conclusions, extend the static typing systems in previous sections to runtime configurations. The auxiliary function `match` in T-Invoc is the same as in T-SyncCall. In T-Object, `fields` constructs a mapping from fields names to field types for a class. A runtime typing context Δ is assumed to contain bindings for runtime identifiers, i.e., object identifiers and future identifiers, to their types. This is reflected in the rules T-State1, T-Cont, T-Future-v, T-Future-bot, and T-Invoc, when considered in conjunction with the rules T-ObjectId and T-FutureId in Figure A.1.

A.3.4 Subject Reduction

Lemma A.3.2 defines a properly typed starting configuration for a given Core ABS program. The configuration consists of a starting object with a process executing the statements in the main block.

$$\frac{\begin{array}{l}(\textsc{T-State1})\\ \Delta \vdash x : T\\ \Delta \vdash v : T\end{array}}{\Delta \vdash_R T\, x\, v\ \mathbf{ok}} \qquad \frac{\begin{array}{l}(\textsc{T-Cont})\\ \Delta \vdash f : \mathbf{Fut}\langle T\rangle\end{array}}{\Delta \vdash_R \mathbf{cont}\,(f);\ \mathbf{ok}} \qquad \frac{\begin{array}{l}(\textsc{T-Future-v})\\ \Delta \vdash f : \mathbf{Fut}\langle T\rangle\\ \Delta \vdash v : T\end{array}}{\Delta \vdash_R \mathsf{fut}\,(f, v)\ \mathbf{ok}} \qquad \frac{\begin{array}{l}(\textsc{T-Future-bot})\\ \Delta \vdash f : \mathbf{Fut}\langle T\rangle\end{array}}{\Delta \vdash_R \mathsf{fut}\,(f, \bot)\ \mathbf{ok}}$$

$$\frac{\begin{array}{l}(\textsc{T-Configurations})\\ \Delta \vdash_R cn\,\mathbf{ok} \quad \Delta \vdash_R cn'\,\mathbf{ok}\end{array}}{\Delta \vdash_R cn\ cn'\,\mathbf{ok}} \qquad \frac{(\textsc{T-Empty})}{\Delta \vdash_R \epsilon\,\mathbf{ok}} \qquad \frac{\begin{array}{l}(\textsc{T-Process-Queue})\\ \Delta \vdash_R q\,\mathbf{ok} \quad \Delta \vdash_R q'\,\mathbf{ok}\end{array}}{\Delta \vdash_R q\ q'\,\mathbf{ok}}$$

$$\frac{(\textsc{T-Empty-Queue})}{\Delta \vdash_R \emptyset\,\mathbf{ok}} \qquad \frac{\begin{array}{l}(\textsc{T-Return})\\ \Delta \vdash e : T \quad \Delta(\,\mathbf{destiny}\,) = \mathbf{Fut}\langle T\rangle\end{array}}{\Delta \vdash_R \mathbf{return}\ e;\ \mathbf{ok}}$$

$$\frac{\begin{array}{l}(\textsc{T-State2})\\ \Delta \vdash_R a\,\mathbf{ok}\\ \Delta \vdash_R a'\,\mathbf{ok}\end{array}}{\Delta \vdash_R a, a'\,\mathbf{ok}} \qquad \frac{\begin{array}{l}(\textsc{T-Invoc})\\ \Delta \vdash f : \mathbf{Fut}\langle T\rangle \quad \Delta \vdash \overline{v} : \overline{T}\\ \mathtt{match}\,(m, \overline{T} \to T, \Delta(o))\end{array}}{\Delta \vdash_R \mathsf{invoc}\,(o, f, m, \overline{v})\,\mathbf{ok}} \qquad \frac{(\textsc{T-Idle})}{\Delta \vdash_R \mathbf{idle}\ \mathbf{ok}}$$

$$\frac{\begin{array}{l}(\textsc{T-Process})\\ \Delta' = \Delta[x_1 \mapsto T_1, .., x_n \mapsto T_n]\\ \Delta' \vdash_R T_1\, x_1\, v_1, .., T_n\, x_n\, v_n\,\mathbf{ok}\\ \Delta' \vdash_R \overline{sp}\,\mathbf{ok}\end{array}}{\Delta \vdash_R \{T_1\, x_1\, v_1, .., T_n\, x_n\, v_n \mid \overline{sp}\}\,\mathbf{ok}} \qquad \frac{\begin{array}{l}(\textsc{T-Object})\\ \mathtt{fields}\,(\Delta(o)) = [x_1 \mapsto T_1, .., x_n \mapsto T_n]\\ \Delta' = \Delta[x_1 \mapsto T_1, .., x_n \mapsto T_n] \quad \Delta' \vdash_R pr\ \mathbf{ok}\\ \Delta' \vdash_R T_1\, x_1\, v_1, .., T_n\, x_n\, v_n\,\mathbf{ok} \quad \Delta' \vdash_R q\ \mathbf{ok}\end{array}}{\Delta \vdash_R \mathsf{ob}\,(o, T_1\, x_1\, v_1\,, .., T_n\, x_n\, v_n, pr, q)\,\mathbf{ok}}$$

Figure A.7: Standard Core ABS runtime typing rules

Lemma A.3.2. *Let $\overline{Dd}$ $\overline{F}$ $\overline{IF}$ $\overline{CL}$ $\{T_1\, x_1; ..; T_n x_n;\ \overline{s}\}$ be a Core ABS program, and let* `value` *be a function that returns the default value for a ground type. If $\Gamma \vdash \overline{Dd}\,\overline{F}\,\overline{IF}\,\overline{CL}$ $\{T_1\, x_1; ..; T_n x_n;\ \overline{s}\}$ for some typing context Γ, then $\Gamma \vdash_R \mathsf{ob}\,(\mathbf{start}, \epsilon, \{T_1\, x_1\, \mathtt{value}(T_1), .., T_n\, x_n\, \mathtt{value}(T_n) \mid \overline{s}\}, \emptyset)\,\mathbf{ok}$.*

Proof. Let $\Gamma' = \Gamma[x_1 \mapsto T_1, .., x_n \mapsto T_n]$. Then, we have

$$\Gamma' \vdash_R T_1\, x_1\, \mathtt{value}(T_1), .., T_n\, x_n\, \mathtt{value}(T_n)\,\mathbf{ok}$$

and hence $\Gamma \vdash_R \mathsf{ob}\,(\mathbf{start}, \epsilon, \{T_1\, x_1\, \mathtt{value}(T_1), .., T_n\, x_n\, \mathtt{value}(T_n) \mid \overline{s}\}, \emptyset)\,\mathbf{ok}$ by T-Object. □

Theorem A.3.3 states that the standard Core ABS semantics preserves well-typing. The theorem implies, among other things, that method invocations in Core ABS cannot go wrong at runtime for type-checked programs; when an object makes a call to a method m using an object identifier o, there always exists an object associated with o, which is an instance of a class where m is defined. Hence, in an execution of a well-typed program beginning from its starting configuration, no object gets stuck trying to execute the **error** process.

Theorem A.3.3 (Subject Reduction)**.** *Let Δ be a typing context and cn a runtime configuration. If $\Delta \vdash_R cn\,\mathbf{ok}$ and $cn \to cn'$, then there exists a typing context Δ' such that $\Delta \subseteq \Delta'$ and $\Delta' \vdash_R cn'\,\mathbf{ok}$.*

Proof. By induction on rule applications [103]. □

Appendix B

ABS-NET Semantics

This appendix defines the network-aware ABS-NET semantics of Core ABS programs, with syntax as defined in Appendix A.

B.1 Runtime Configurations

The runtime syntax of ABS-NET is shown in Table B.1. A network *net* consists of nodes and arcs, composed with the whitespace operator, with ϵ the empty network. In a node $\mathsf{nd}\,(u, \tau)$, u is a node identifier (assumed globally unique) and τ is a routing table, used to route object-related messages in the proper direction. In an arc $\mathsf{ar}\,(u, Q, u')$, representing a unidirectional link from u to u', Q is a FIFO-ordered queue of messages *msg*.

cn	$::=$	$\epsilon \mid object \mid cn\ cn$	net	$::=$	$\epsilon \mid node \mid arc \mid net\ net$
$object$	$::=$	$\mathsf{ob}\,(o, a, pr, q, Q_{in}, Q_{out}, \Sigma)$	$node$	$::=$	$\mathsf{nd}\,(u, \tau)$
a, l	$::=$	$\epsilon \mid T\,x\,v \mid a, a$	arc	$::=$	$\mathsf{ar}\,(u, Q, u)$
$process$	$::=$	$\{l \mid \overline{sp}\} \mid \mathbf{error}$	q	$::=$	$\emptyset \mid process \mid q\ q$
msg	$::=$	$\textsc{Call}(o, o, f, m, \overline{v}) \mid \textsc{Table}(\tau)$	sp	$::=$	$\mathbf{return}\ e\,; \mid \mathbf{cont}\,(f)\,; \mid s$
		$\mid \textsc{Future}(o, f, v) \mid \textsc{Object}(object)$	pr	$::=$	$process \mid \mathbf{idle}$

Table B.1: ABS-NET runtime syntax

An object configuration *cn* consists of objects, composed by the whitespace operator, again with ϵ as the empty configuration. In objects $\mathsf{ob}\,(o, a, pr, q, Q_{in}, Q_{out}, \Sigma)$, a is a store for its field types and values, pr an active process, q a pool of suspended processes, Q_{in} and Q_{out} input and output queues, and Σ a structure for storing resolved future values and obligations to send future values. Stores, active processes, process statements and process pools are defined in the same way as in the standard semantics in Section A.3.

The method invocations and futures present in runtime configurations of the standard semantics can intuitively be said to have been replaced in ABS-NET with call and future messages, which are transported from node to node via arcs. A call message $\textsc{Call}(o, o', f, m, \overline{v})$ consists of the identifier o of the destination object, the identifier o' of the sender object, the associated future identifier f, method name m and argument list $\overline{v}$. A future value message $\textsc{Future}(o, f, v)$ has the identifier of the destination object o, the future identifier f and the associated resolved value v.

Table and object messages, on the other hand, have no equivalent in the standard semantics. A table message $\textsc{Table}(\tau)$ is used to pass a routing table τ from one node to another, allowing local routes to be updated with new information from a neighbor. An object message $\textsc{Object}(\mathit{object})$ contains a complete runtime object *object* and is what facilitates object mobility for network-adaptable process execution.

B.2 Reduction System for Guard Expressions

The standard reduction system for functional expressions, given in Figure A.3, is carried over to the ABS-NET semantics unchanged. The rules for guard evaluation in ABS-NET, given in Figure B.1, are different, however. The rules highlight how the structure Σ, instead of a configuration cn, is queried for the resolved values of futures. Given a future identifier f, `valof` (f, Σ) returns a val, i.e., either a value v or $\bot$. If the result is a value, the future has been recorded as resolved locally. Note that in the semantics generally, the fact that a future is unresolved locally does not mean the associated method invocation is unfinished—there may be a future message incoming.

$$\textsc{(Net-RedBoolGuard)}\quad \frac{\sigma \vdash b \rightsquigarrow \sigma \vdash b'}{\sigma, \Sigma \vdash b \rightsquigarrow \sigma, \Sigma \vdash b'}$$

$$\textsc{(Net-RedReplyGuard1)}\quad \frac{\sigma \vdash e \rightsquigarrow \sigma \vdash f \quad \texttt{valof}\,(f, \Sigma) \neq \bot}{\sigma, \Sigma \vdash e? \rightsquigarrow \sigma, \Sigma \vdash \textbf{True}}$$

$$\textsc{(Net-RedReplyGuard2)}\quad \frac{\sigma \vdash e \rightsquigarrow \sigma \vdash f \quad \texttt{valof}\,(f, \Sigma) = \bot}{\sigma, \Sigma \vdash e? \rightsquigarrow \sigma, \Sigma \vdash \textbf{False}}$$

$$\textsc{(Net-RedGuards)}\quad \frac{\sigma, \Sigma \vdash g_1 \rightsquigarrow \sigma, \Sigma \vdash g_1' \quad \sigma, \Sigma \vdash g_2 \rightsquigarrow \sigma, \Sigma \vdash g_2'}{\sigma, \Sigma \vdash g_1 \wedge g_2 \rightsquigarrow \sigma, \Sigma \vdash g_1' \wedge g_2'}$$

Figure B.1: ABS-NET reduction rules for guard expressions

The boolean result of evaluating a guard expression g with respect to a structure Σ and a store a in ABS-NET is written as $[\![g]\!]_a^\Sigma$. As in the standard semantics, the ground term result of evaluating an expression e with respect to a store a is written as $[\![e]\!]_a$.

B.3 Message Queues

The nature of a FIFO queue Q of messages is specified through three auxiliary functions: `enqueue`, `dequeue` and `first`. `enqueue`(Q, msg) returns the queue that results when the message *msg* is added to the back of Q. If Q is non-empty, `first`(Q) returns the message at the front of Q, and `dequeue`(Q) returns the queue that results when the front message is removed. For brevity, `enqueue`$(Q, \mathit{msg}) = Q'$ is defined as a relation $Q \xrightarrow{\texttt{enqueue}\,(\mathit{msg})} Q'$, while the conjunction that `first`$(Q) = \mathit{msg}$ and `dequeue`$(Q) = Q'$ is defined as $Q \xrightarrow{\texttt{dequeue}\,(\mathit{msg})} Q'$. $()$ is the empty queue.

Appropriate encodings of queues in implementations vary significantly depending on the application and environment. For example, when using TCP sockets as arcs, the socket library used will determine the queue characteristics. For the purpose of formal analysis, a simple algebraic list-like encoding suffices.

B.4 Routing Tables

The nature of a routing table is specified through the four auxiliary functions `update`, `next`, `register` and `replace`, and an infix operator $\in$. The auxiliary function `update` takes three arguments: the routing table τ of the current node, the node identifier u' of the adjacent node, and the routing table τ' of the adjacent node. The function returns a routing table τ'', which incorporates the routes from τ' into τ if appropriate, with the constraint that all such routes must go through the node u'. For brevity, `update`$(\tau, u', \tau') = \tau''$ is defined as a relation $\tau \xrightarrow{\texttt{update}\,(\tau',u')} \tau''$. The auxiliary function `next` takes three arguments: the routing table τ of the current node, the object identifier o' of the node we want the next hop for, and the default hop u, which is the identifier of the current node. The function returns the node identifier u' which is the next hop of o' according to the table. The auxiliary function `register` takes four arguments: the routing table τ of the current node, the object identifier o' of the object we want to add a route for, the node identifier u of a neighbor node (usually self) which is the next hop, and a non-negative integer k for the distance to the object (in all instances in the rules, it is 0). The function returns a routing table τ' which incorporates the new route. For brevity, `register`$(\tau, o', u, k) = \tau'$ is defined as a relation $\tau \xrightarrow{\texttt{register}\,(o',u,k)} \tau'$. The auxiliary function `replace` takes four arguments (of the same kind as `register`): the routing table τ of the current node, the object identifier o' of the object we want to replace the route for, the node identifier u of a neighbor node which is the next hop, and a natural number k for the distance to the object. The function returns a routing table τ' which has removed any existing routes for o' and added the route given. For brevity, `replace`$(\tau, o', u, k) = \tau'$ is defined as a relation $\tau \xrightarrow{\texttt{replace}\,(o,u,k)} \tau'$. The claim $o \in \tau$, with a node identifier u given by the context, means that, according to τ, the object with identifier o is located on the node u.

One example of a relatively simple encoding of a routing table, which can be enough for some implementations, is as a finite map from object identifiers to sets of tuples of node identifiers and distances. A binding in a routing table for the identifier o to the set $\{(u, 2), (u', 3)\}$ then represents that the next hop for reaching the object o is either in the direction of u, for a total distance of 2 hops, or in the direction of u', for a total distance of 3 hops. Of course, due to object mobility, the routes may not be accurate, but are able to reflect the last known information.

B.5 Operational Semantics of Networks

The global state in ABS-NET consists of a pair $\{net\}\,\{cn\}$. The network part and the object part of the global state can evolve jointly by performing synchronized labeled transitions, but also separately without exchanging information. The rules for such synchronization and separate evolution are shown in Figure B.2. A label α is either $\texttt{mv}(\mathit{object})$ (moving an object), $\texttt{rg}(o, o')$ (registering a new object identifier), or $\texttt{tr}(o, \mathit{msg})$ (transporting a message). Intuitively, a label with an overline means that information is outgoing or being sent, while a label without overline means information is incoming or being received. Like in the standard semantics, an execution is a sequence global states with valid rule transitions between every adjacent pair.

$$\text{(Net-Red)}\ \frac{\{net\} \to \{net'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn\}} \qquad \text{(Cn-Red)}\ \frac{\{cn\} \to \{cn'\}}{\{net\}\,\{cn\} \to \{net\}\,\{cn'\}}$$

$$\text{(Cn-Out-Net-In-Red)}\ \frac{\{cn\} \xrightarrow{\overline{\alpha}} \{cn'\} \quad \{net\} \xrightarrow{\alpha} \{net'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn'\}} \qquad \text{(Net-Out-Cn-In-Red)}\ \frac{\{net\} \xrightarrow{\overline{\alpha}} \{net'\} \quad \{cn\} \xrightarrow{\alpha} \{cn'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn'\}}$$

Figure B.2: ABS-NET reduction rules connecting objects and networks

The reduction rules for networks are shown in Figure B.3. The auxiliary function `id`, used in the rule Net-Object-Recv-Out, takes a runtime object as argument and returns its identifier. The function `dest`, used in the rules Net-Msg-Recv-Out, Net-Msg-Send-In and Net-Route-Further, is defined only for Call and Future messages; it returns their first object identifier, which is the identifier of the intended recipient object (destination).

For proper progress in execution, we assume networks are such that (1) there are no dangling arcs referencing non-existent nodes, (2) for every arc between nodes there is an arc in the opposite direction, and (3) every node comes with a self-loop arc, i.e., an arc going from and to the node. Self-loop arcs are important for two reasons. First, it allows us to use the same rules for message passing in both the case where the sender object is at a different node from the receiver object, and where the sender is at the same node as the receiver. Once a message has been put in the self-loop queue, it intuitively appears as if it came from some other node when applying the rule Net-Msg-Recv-Out. Second, it may not always be the case that there is a route (next hop) to the recipient of a message, because routing tables may not have stabilized. Such a message must be dealt with somehow when there are other important messages pending in that queue after the message. Hence, it is put in the self-loop queue, i.e., the default next hop of an object-addressed message is the node itself.

$$\textsc{(Net-Table-Send)}\quad \frac{\begin{array}{l} u' \neq u \\ Q \xrightarrow{\mathtt{enqueue}\,(\textsc{Table}\,(\tau))} Q' \end{array}}{\begin{array}{l} \mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,\,Q,\,u') \\ \to \mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,\,Q',\,u') \end{array}}$$

$$\textsc{(Net-Table-Recv)}\quad \frac{\begin{array}{l} Q \xrightarrow{\mathtt{dequeue}\,(\textsc{Table}\,(\tau'))} Q' \\ \tau \xrightarrow{\mathtt{update}\,(\tau',u')} \tau'' \end{array}}{\begin{array}{l} \mathsf{ar}\,(u',\,Q,\,u)\;\mathsf{nd}\,(u,\tau) \\ \to \mathsf{ar}\,(u',\,Q',\,u)\;\mathsf{nd}\,(u,\tau'') \end{array}}$$

$$\textsc{(Net-Msg-Recv-Out)}\quad \frac{\begin{array}{l} Q \xrightarrow{\mathtt{dequeue}\,(msg)} Q' \\ \mathtt{dest}\,(msg) = o \quad o \in \tau \end{array}}{\begin{array}{l} \mathsf{ar}\,(u',\,Q,\,u)\;\mathsf{nd}\,(u,\tau) \\ \overset{\mathtt{tr}\,(o,msg)}{\to}\; \mathsf{ar}\,(u',\,Q',\,u)\;\mathsf{nd}\,(u,\tau) \end{array}}$$

$$\textsc{(Net-Msg-Send-In)}\quad \frac{\begin{array}{l} o \in \tau \quad \mathtt{dest}\,(msg) = o' \\ \mathtt{next}\,(\tau,\,o',\,u) = u' \\ Q \xrightarrow{\mathtt{enqueue}\,(msg)} Q' \end{array}}{\begin{array}{l} \mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,\,Q,\,u') \\ \overset{\mathtt{tr}\,(o,msg)}{\to}\; \mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,\,Q',\,u') \end{array}}$$

$$\textsc{(Net-Route-Further)}\quad \frac{\begin{array}{l} Q_1 \xrightarrow{\mathtt{dequeue}\,(msg)} Q_1' \quad \mathtt{dest}\,(msg) = o \quad o \notin \tau \\ \mathtt{next}\,(\tau,\,o,\,u) = u'' \quad Q_2 \xrightarrow{\mathtt{enqueue}\,(msg)} Q_2' \end{array}}{\begin{array}{l} \mathsf{ar}\,(u',\,Q_1,\,u)\;\mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,\,Q_2,\,u'') \\ \to \mathsf{ar}\,(u',\,Q_1',\,u)\;\mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,\,Q_2',\,u'') \end{array}}$$

$$\textsc{(Net-Object-Send-In)}\quad \frac{\begin{array}{l} o \in \tau \quad u' \neq u \\ \tau \xrightarrow{\mathtt{replace}\,(o,u',1)} \tau' \\ Q \xrightarrow{\mathtt{enqueue}\,(\textsc{Object}\,(object))} Q' \end{array}}{\begin{array}{l} \mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,\,Q,\,u') \\ \overset{\mathtt{mv}\,(object)}{\to}\; \mathsf{nd}\,(u,\tau')\;\mathsf{ar}\,(u,\,Q',\,u') \end{array}}$$

$$\textsc{(Net-Object-Recv-Out)}\quad \frac{\begin{array}{l} \mathtt{id}\,(object) = o \\ Q \xrightarrow{\mathtt{dequeue}\,(\textsc{Object}\,(object))} Q' \\ \tau \xrightarrow{\mathtt{replace}\,(o,u,0)} \tau' \end{array}}{\begin{array}{l} \mathsf{ar}\,(u',\,Q,\,u)\;\mathsf{nd}\,(u,\tau) \\ \overset{\mathtt{mv}\,(object)}{\to}\; \mathsf{ar}\,(u',\,Q',\,u)\;\mathsf{nd}\,(u,\tau') \end{array}}$$

$$\textsc{(Net-New-Object-In)}\quad \frac{\mathtt{fresh}\,(o') \quad o \in \tau \quad \tau \xrightarrow{\mathtt{register}\,(o',u,0)} \tau'}{\mathsf{nd}\,(u,\tau) \overset{\mathtt{rg}\,(o,o')}{\to} \mathsf{nd}\,(u,\tau')}$$

Figure B.3: ABS-NET node controller reduction rules

The property of an object being located on a node is represented indirectly through the rules, not explicitly in runtime configurations. The labeled transition rules Net-Msg-Recv-Out, Net-Msg-Send-In, Net-Object-Send-In and Net-New-Object-In, where a node exchanges information with an object, all use the premise $o \in \tau$ to restrict actions to pertain to node-local objects.

B.6 Future Value Distribution

In the ABS-NET semantics, future values are transmitted through messages to the objects which need them, as opposed to the centralized future access in the standard semantics. However, as described by Henrio et al. [90], there are several fundamentally different ways of propagating futures to objects, with different trade-offs in performance and resource usage. The ABS-NET semantics uses what Henrio et al. refer to as an eager forward-based strategy, where an object o that shares a future identifier f with another object o' is obligated to forward the value of f to

o' when this becomes possible. This strategy is relatively easy to implement and distributes the load of messaging related to futures over many objects, which in balanced object-node allocations translates to many nodes.

A number of auxiliary functions in the ABS-NET reduction rules for objects take a structure Σ as input and either retrieve data from it or produce a modified structure in order to accomplish future forwarding. They are reminiscent of the operations modelled by Henrio et al., but have several properties specific to the ABS setting. The function `recsof` takes a future identifier f and a structure as input, and returns a set of object identifiers; intuitively, this set contains the identifiers of the objects which are the intended recipients of the value of f. The function `sendfuts` takes a set of future identifiers, an object identifier, and a structure, and returns another structure, interpreted as the given structure updated with obligations to forward the values of all indicated futures to the indicated object. For all structures Σ and all future identifiers $f \in \{f_1, \ldots, f_n\}$, it holds that $o \in \texttt{recsof}\,(f, \texttt{sendfuts}\,(\{f_1, \ldots, f_n\}, o, \Sigma))$. The function `clrrec`, which takes an object identifier o, a future identifier f and a structure Σ, returns an updated structure Σ' where o has been cleared from the recipient set of f, i.e., $o \notin \texttt{recsof}\,(f, \Sigma')$. The function `regfuts` takes a set of future identifiers and a structure Σ, and returns an updated structure Σ' where the given futures are registered, meaning that the futures are associated through Σ' with a *val* term and a set of object identifiers. If a future f has no such associations in Σ, it is the case that $\texttt{valof}\,(f, \Sigma') = \bot$ and $\texttt{recsof}\,(f, \Sigma') = \emptyset$; otherwise, the associations are the same as in Σ. Finally, the function `resfut` takes a future identifier f, a value v and a structure Σ, and returns an updated structure Σ' where v is recorded as the resolved value for f, i.e., $\texttt{valof}\,(f, \Sigma') = v$.

Most straightforwardly, a structure Σ can be encoded as a pair $\langle M_{\text{VAL}}, M_{\text{REC}} \rangle$, where M_{VAL} is a finite map from future identifiers to *val* terms, and M_{REC} is a finite map from future identifiers to sets of object identifiers. The auxiliary functions are then defined as map operations, e.g., $\texttt{valof}(f, \langle M_{\text{VAL}}, M_{\text{REC}} \rangle) \triangleq M_{\text{VAL}}(f)$ and $\texttt{recsof}(f, \langle M_{\text{VAL}}, M_{\text{REC}} \rangle) \triangleq M_{\text{REC}}(f)$.

B.7 Operational Semantics of Concurrent Objects

The labeled rules for object configurations are shown in Figure B.4. The `init` and `atts` auxiliary functions constructs the initial task of the object as given in the corresponding class definition, and initializes variables based on given arguments, respectively, and are unchanged from Core ABS. The function `futsof` returns the set of all future identifiers in a given value, if necessary by recursively examining algebraic datatype terms. $[\,]$ is the empty structure of future values and obligations.

For object creation, both the network rule Net-New-Object-In and object rule ABS-New-Object-Out need to be involved. When such a synchronized transition has taken place, the new object has been properly added to the interpreter layer, and its globally unique identifier registered on the node of the object that spawned it. Given

$$\frac{(\text{ABS-OBJECT-SEND-OUT})}{\{object\ cn\} \overset{\overline{\texttt{mv}\,(object)}}{\to} \{cn\}} \qquad \frac{(\text{ABS-OBJECT-RECV-IN})}{\{cn\} \overset{\texttt{mv}\,(object)}{\to} \{object\ cn\}}$$

$$\frac{\begin{array}{l}(\text{ABS-MSG-SEND-OUT})\\ Q_{out} \xrightarrow{\texttt{dequeue}\,(msg)} Q'_{out}\end{array}}{\begin{array}{l}\mathsf{ob}\,(o, a, pr, q, Q_{in}, Q_{out}, \Sigma)\\ \overset{\overline{\texttt{tr}\,(o, msg)}}{\to} \mathsf{ob}\,(o, a, pr, q, Q_{in}, Q'_{out}, \Sigma)\end{array}} \qquad \frac{\begin{array}{l}(\text{ABS-MSG-RECV-IN})\\ Q_{in} \xrightarrow{\texttt{enqueue}\,(msg)} Q'_{in}\end{array}}{\begin{array}{l}\mathsf{ob}\,(o, a, pr, q, Q_{in}, Q_{out}, \Sigma)\\ \overset{\texttt{tr}\,(o, msg)}{\to} \mathsf{ob}\,(o, a, pr, q, Q'_{in}, Q_{out}, \Sigma)\end{array}}$$

$$\frac{\begin{array}{l}(\text{ABS-NEW-OBJECT-OUT})\\ \texttt{init}\,(C) = process \quad [\![\overline{e}]\!]_{a \circ l} = \overline{v} \quad \texttt{sendfuts}\,(\texttt{futsof}\,(\overline{v}), o', \Sigma) = \Sigma' \\ \texttt{atts}\,(C, \overline{v}, o') = a' \quad \texttt{regfuts}\,(\texttt{futsof}\,(\overline{v}), [\,]) = \Sigma''\end{array}}{\begin{array}{l}\mathsf{ob}\,(o, a, \{l \mid x = \mathbf{new}\ C(\overline{e});\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)\\ \overset{\overline{\texttt{rg}\,(o, o')}}{\to} \mathsf{ob}\,(o, a, \{l \mid x = o';\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma')\,\mathsf{ob}\,(o', a', \mathbf{idle}, process, (), (), \Sigma'')\end{array}}$$

Figure B.4: ABS-NET object reduction rules for node controller interaction

that we abstract from details on marshalling and pass object states directly in messages, the rules for object mobility, ABS-OBJECT-SEND-OUT and ABS-OBJECT-RECV-IN, which interact with the rules NET-OBJECT-SEND-IN and NET-OBJECT-RECV-OUT above, are straightforward. The rules ABS-MSG-SEND-OUT and ABS-MSG-RECV-IN for passing messages back and forth with the node controller are uncomplicated since eligible messages have been put in the out queue of the object.

The remaining rules for object transitions, that do not involve information exchange with a node via a label, are given in Figure B.5 and Figure B.6. In ABS-ACTIVATE, the auxiliary function `select` takes two parameters: the local pool of processes q and the object store a, skipping the configuration parameter in the corresponding function in the standard semantics. The reason is that scheduling in ABS-NET is assumed to take only local information into account. In ABS-REM-SYNC-CALL, the premise $o' \neq o$ is an addition when compared to the standard semantics counterpart rule. This premise is used to preclude synchronous self calls from being dispatched asynchronously, causing a deadlock. In the Core ABS semantics, such deadlocks are ruled out by the presence of the other runtime object in the rule's left-hand side. In ABS-NET, the intent is for all rules to be implementable directly on a single node, which implies it is not possible to depend on both the caller and callee objects being present locally.

An ABS-NET global starting state is given by a network configuration having the properties described above, and an object configuration containing a single starting object as in Lemma A.3.2, extended with empty queues and an empty structure of futures and obligations.

$$\frac{(\text{ABS-}\textsc{Skip})}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \textbf{skip}\,;\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)\\ \rightarrow \mathsf{ob}\,(o, a, \{l \mid \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)\end{array}}$$

$$\frac{\begin{array}{c}(\text{ABS-}\textsc{Assign-Local})\\ x \in \mathtt{dom}\,(l) \quad [\![e]\!]_{a \circ l} = v\end{array}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid x = e;\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)\\ \rightarrow \mathsf{ob}\,(o, a, \{l[x \mapsto v] \mid \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)\end{array}}$$

$$\frac{\begin{array}{c}(\text{ABS-}\textsc{Assign-Field})\\ x \in \mathtt{dom}\,(a) \quad [\![e]\!]_{a \circ l} = v\end{array}}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid x = e;\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)\\ \rightarrow \mathsf{ob}\,(o, a[x \mapsto v], \{l \mid \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)\end{array}}$$

$$\frac{(\text{ABS-}\textsc{Suspend})}{\begin{array}{c}\mathsf{ob}\,(o, a, \{l \mid \textbf{suspend}\,;\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)\\ \rightarrow \mathsf{ob}\,(o, a, \textbf{idle}, q \cup \{l \mid \overline{sp}\}, Q_{in}, Q_{out}, \Sigma)\end{array}}$$

$$\frac{\begin{array}{l}(\text{ABS-}\textsc{Cond-True})\\ [\![b]\!]_{a \circ l} = \textbf{True}\end{array}}{\mathsf{ob}\,(o, a, \{l \mid \textbf{if}\ b\{\overline{s}\}\ [\textbf{else}\ \{\overline{s}'\}]\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \rightarrow \mathsf{ob}\,(o, a, \{l \mid \overline{s}\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

$$\frac{\begin{array}{l}(\text{ABS-}\textsc{Cond-False})\\ [\![b]\!]_{a \circ l} = \textbf{False}\end{array}}{\mathsf{ob}\,(o, a, \{l \mid \textbf{if}\ b\{\overline{s}\}\ [\textbf{else}\ \{\overline{s}'\}]\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \rightarrow \mathsf{ob}\,(o, a, \{l \mid [\overline{s}']\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

$$\frac{\begin{array}{l}(\text{ABS-}\textsc{Await-True})\\ [\![g]\!]^{\Sigma}_{a \circ l} = \textbf{True}\end{array}}{\mathsf{ob}\,(o, a, \{l \mid \textbf{await}\ g;\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \rightarrow \mathsf{ob}\,(o, a, \{l \mid \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

$$\frac{\begin{array}{l}(\text{ABS-}\textsc{Await-False})\\ [\![g]\!]^{\Sigma}_{a \circ l} = \textbf{False}\end{array}}{\mathsf{ob}\,(o, a, \{l \mid \textbf{await}\ g;\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \rightarrow \mathsf{ob}\,(o, a, \{l \mid \textbf{suspend}\,;\, \textbf{await}\ g;\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

$$\frac{\begin{array}{l}(\text{ABS-}\textsc{Async-Call-Send})\\ \mathtt{regfuts}\,(\{f\}, \mathtt{sendfuts}\,(\mathtt{futsof}\,(\overline{v}), o', \Sigma)) = \Sigma'\\ [\![e]\!]_{a \circ l} = o' \quad [\![\overline{e}]\!]_{a \circ l} = \overline{v} \quad \mathtt{fresh}\,(f) \quad Q_{out} \xrightarrow{\mathsf{enqueue}\,(\textsc{Call}\,(o', o, f, m, \overline{v}))} Q'_{out}\end{array}}{\mathsf{ob}\,(o, a, \{l \mid x = e!m(\overline{e});\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \rightarrow \mathsf{ob}\,(o, a, \{l \mid x = f;\, \overline{sp}\}, q, Q_{in}, Q'_{out}, \Sigma')}$$

$$\frac{\begin{array}{l}(\text{ABS-}\textsc{Async-Call-Recv})\\ Q_{in} \xrightarrow{\mathsf{dequeue}\,(\textsc{Call}\,(o, o', f, m, \overline{v}))} Q'_{in} \quad \mathtt{bind}\,(o, f, m, \overline{v}, \mathtt{class}\,(o)) = \mathit{process}\\ \mathtt{sendfuts}\,(\{f\}, o', \mathtt{regfuts}\,(\mathtt{futsof}\,(\overline{v}) \cup \{f\}, \Sigma)) = \Sigma'\end{array}}{\begin{array}{c}\mathsf{ob}\,(o, a, pr, q, Q_{in}, Q_{out}, \Sigma)\\ \rightarrow \mathsf{ob}\,(o, a, pr, q \cup \mathit{process}, Q'_{in}, Q_{out}, \Sigma')\end{array}}$$

Figure B.5: ABS-NET object reduction rules, part 1

(ABS-FUTURE-RECV)
$$\frac{Q_{in} \xrightarrow{\texttt{dequeue}\,(\textsc{Future}\,(o,f,v))} Q'_{in} \quad \texttt{resfut}\,(f, v, \Sigma) = \Sigma'}{\mathsf{ob}\,(o, a, pr, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, pr, q, Q'_{in}, Q_{out}, \Sigma')}$$

(ABS-FUTURE-SEND)
$$\frac{Q_{out} \xrightarrow{\texttt{enqueue}\,(\textsc{Future}\,(o',f,v))} Q'_{out} \quad \texttt{valof}\,(f, \Sigma) = v \quad o' \in \texttt{recsof}\,(f, \Sigma) \quad \texttt{sendfuts}\,(\texttt{futsof}\,(v), o', \texttt{clrrec}\,(o', f, \Sigma)) = \Sigma'}{\mathsf{ob}\,(o, a, pr, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, pr, q, Q_{in}, Q'_{out}, \Sigma')}$$

(ABS-READ-FUT)
$$\frac{[\![e]\!]_{a \circ l} = f \quad \texttt{valof}\,(f, \Sigma) = v}{\mathsf{ob}\,(o, a, \{l \mid x = e.\,\mathbf{get};\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, \{l \mid x = v;\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

(ABS-IDLE)
$$\frac{}{\mathsf{ob}\,(o, a, \{l \mid \}, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, \mathbf{idle}, q, Q_{in}, Q_{out}, \Sigma)}$$

(ABS-ACTIVATE)
$$\frac{\texttt{select}\,(q, a) = process}{\mathsf{ob}\,(o, a, \mathbf{idle}, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, process, q \setminus process, Q_{in}, Q_{out}, \Sigma)}$$

(ABS-RETURN)
$$\frac{[\![e]\!]_{a \circ l} = v \quad l(\,\mathbf{destiny}\,) = f \quad \texttt{resfut}\,(f, v, \Sigma) = \Sigma'}{\mathsf{ob}\,(o, a, \{l \mid \mathbf{return}\, e;\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, \{l \mid \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma')}$$

(ABS-SELF-SYNC-RETURN-SCHED)
$$\frac{l'(\,\mathbf{destiny}\,) = f}{\mathsf{ob}\,(o, a, \{l \mid \mathbf{cont}\,(f); \}, q \cup \{l' \mid \overline{sp}\}, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, \{l' \mid \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

(ABS-SELF-SYNC-CALL)
$$\frac{l(\,\mathbf{destiny}\,) = f' \quad [\![e]\!]_{a \circ l} = o \quad [\![\overline{e}]\!]_{a \circ l} = \overline{v} \quad \texttt{fresh}\,(f) \quad \texttt{regfuts}\,(\{f\}, \Sigma) = \Sigma' \quad \texttt{bind}\,(o, f, m, \overline{v}, \texttt{class}\,(o)) = \{l' \mid \overline{sp}'\}}{\mathsf{ob}\,(o, a, \{l \mid x = e.m(\overline{e});\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, \{l' \mid \overline{sp}'\ \mathbf{cont}\,(f'); \}, q \cup \{l \mid x = f.\,\mathbf{get};\ \overline{sp}\}, Q_{in}, Q_{out}, \Sigma')}$$

(ABS-REM-SYNC-CALL)
$$\frac{[\![e]\!]_{a \circ l} = o' \quad o' \neq o \quad \texttt{fresh}\,(y) \quad \texttt{returns}\,(\,\texttt{class}\,(o'), m) = T}{\mathsf{ob}\,(o, a, \{l \mid x = e.m(\overline{e});\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, \{l, \mathbf{Fut}\,\langle T\rangle\, y\, \mathbf{null} \mid y = o'!m(\overline{e});\ x = y.\,\mathbf{get};\ \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

(ABS-WHILE-TRUE)
$$\frac{[\![b]\!]_{a \circ l} = \mathbf{True}}{\mathsf{ob}\,(o, a, \{l \mid \mathbf{while}\, b\, \{\overline{s}\}\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, \{l \mid \overline{s}\ \mathbf{while}\, b\{\overline{s}\}\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

(ABS-WHILE-FALSE)
$$\frac{[\![b]\!]_{a \circ l} = \mathbf{False}}{\mathsf{ob}\,(o, a, \{l \mid \mathbf{while}\, b\, \{\overline{s}\}\, \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma) \to \mathsf{ob}\,(o, a, \{l \mid \overline{sp}\}, q, Q_{in}, Q_{out}, \Sigma)}$$

Figure B.6: ABS-NET object reduction rules, part 2

Appendix C

Shutdown Protocol Definition

This appendix lists the state transitions of a node in the shutdown protocol.

C.1 Functions, Constants, Predicates, and Statements

All set-related functions and constants used in the definition are listed in Table C.1. All functions and constants related to routing tables are listed in Table C.2. Finally, all special constants, predicates, and statements are given in Table C.3.

`{}`	the empty set
`in(u,set)`	set membership status of `u` in `set`
`union(set1,set2)`	set union of `set1` and `set2`
`diff(set1,set2)`	set difference of `set1` and `set2`
`intersect(set1,set2)`	set intersection of `set1` and `set2`
`singleton(u)`	the set consisting of only the element `u`

Table C.1: Set-related functions and constants

`[]`	the empty routing table
`next(o,table,set)`	node identifier of next hop to reach `o` according to routing table `table`, which is in the set `set`
`remove(u,table)`	`table`, excluding all entries with `u` as next hop
`update(table1,u,table2)`	`table1`, with routes from `table2`, with next hop `u`
`register(o, u, table, i)`	`table`, with `u` as the next hop to reach `o`, for `i` hops in total

Table C.2: Routing table related functions and constants

self	a node's own identifier
init	true when a node is first initialized, false otherwise
exists local objects	true if there is some local object, false otherwise
receive `msg` **from** `u`	true when the first message in the link from `u` is `msg`, false otherwise; discards message if used
send `msg` **to** `u;`	statement, which, when executed, results in dispatching `msg` to `u`
shutdown;	statement, which, when executed, shuts a node down
loop empty	true if a node's self-loop link is empty; false otherwise
`u` **shuts down**	true if the node `u` shuts down; false otherwise
deliver `Message(o, p);`	statement, which, when executed, delivers a message with payload `p` to local object `o`
dispatch `Message(o, p)`	true if a local object dispatched a message to `o` with payload `p`; false otherwise
`obj` **local object**	true if `obj` is a local object, false otherwise
delete object `obj;`	statement, which, when executed, deletes the local object `obj`
`id(obj)`	the identifier of the object `obj`

Table C.3: Special constants, statements, and predicates

C.2 Node Actions

```
upon init
do {
  table := [];
  unblocked := {};
  blocked := {};
  state := IDLE;
  recd_ready = {};
  sent_prepare = {};
  sent_abort = {};
  sent_shutdown = {};
  recd_ack = {};
}

/* SINGLETON SHUTDOWN */

upon state = IDLE
 /\ union(unblocked, blocked) = {}
 /\ ~ exists local objects
 /\ loop empty
do {
  shutdown;
}
```

```
/* IDLE STATE */

upon state = IDLE
 /\ receive Notify from u
 /\ ~ in(u, union(unblocked, blocked))
do {
  unblocked := union(unblocked, singleton(u));
  send Notify to u;
}

upon state = IDLE
 /\ receive Notify from u
 /\ in(u, blocked)
do {
  blocked := diff(blocked, singleton(u));
  unblocked := union(unblocked, singleton(u));
}

upon state = IDLE
 /\ receive Prepare from u
do {
  unblocked := diff(unblocked, singleton(u));
  blocked := union(blocked, singleton(u));
  send Ready to u;
}

upon state = IDLE
 /\ receive Abort from u
do {
  blocked := diff(blocked, singleton(u));
  unblocked := union(unblocked, singleton(u));
}

upon state = IDLE
 /\ receive Shutdown from u
do {
  send Ack to u;
}

upon state = IDLE
 /\ u shuts down
do {
  table := remove(u, table);
  blocked := diff(blocked, singleton(u));
}

upon state = IDLE
 /\ u =/= self
 /\ ~ in (u, union(unblocked, blocked))
do {
  blocked := union(blocked, singleton(u));
  send Notify to u;
}
```

```
upon state = IDLE
 /\ unblocked =/= {}
 /\ blocked = {}
do {
  state := TRANSACT;
}

/* TRANSACT STATE */

upon state = TRANSACT
 /\ in(u, union(unblocked, blocked))
 /\ ~ in(u, sent_prepare)
do {
  sent_prepare := union(sent_prepare, singleton(u));
  send Prepare to u;
}

upon state = TRANSACT
 /\ receive Ready from u
do {
  recd_ready := union(recd_ready, singleton(u));
}

upon state = TRANSACT
 /\ receive Prepare from u
 /\ u < self
do {
  unblocked := diff(unblocked, singleton(u));
  blocked := union(blocked, singleton(u));
  state := ABORT;
  send Ready to u;
}

upon state = TRANSACT
 /\ receive Prepare from u
 /\ ~ u < self
do {
  if ~ in(u, sent_prepare) {
    sent_prepare := union(sent_prepare, singleton(u));
    send Prepare to u;
  }
}

upon state = TRANSACT
 /\ union(unblocked, blocked) = recd_ready
do {
  state := CLEAR;
}
```

```
/* ABORT STATE */

upon state = ABORT
 /\ receive Ready from u
do {
  recd_ready := union(recd_ready, singleton(u));
}

upon state = ABORT
 /\ receive Prepare from u
 /\ in(u, sent_prepare)
 /\ ~ in(u, recd_ready)
do {
  if u < self {
    unblocked := diff(unblocked, singleton(u));
    blocked := union(blocked, singleton(u));
    send Ready to u;
  }
}

upon state = ABORT
 /\ in(u, unblocked)
 /\ in(u, recd_ready)
 /\ ~ in(u, sent_abort)
do {
  sent_abort := union(sent_abort, singleton(u));
  send Abort to u;
}

upon state = ABORT
 /\ intersect(unblocked, sent_prepare) = sent_abort
do {
  recd_ready := {};
  sent_prepare := {};
  sent_abort := {};
  state := IDLE;
}

/* CLEAR STATE */

upon state = CLEAR
 /\ ~ exists local objects
 /\ loop empty
do {
  state := SHUTDOWN;
}

/* SHUTDOWN STATE */

upon state = SHUTDOWN
 /\ receive Ack from u
do {
  recd_ack := union(recd_ack, singleton(u));
}
```

```
upon state = SHUTDOWN
 /\ in(u, union(unblocked, blocked))
 /\ ~ in(u, sent_shutdown)
do {
  sent_shutdown := union(sent_shutdown, singleton(u));
  send Shutdown to u;
}

upon state = SHUTDOWN
 /\ union(unblocked, blocked) = recd_ack
do {
  shutdown;
}

/* OBJECTS AND TABLES */

upon receive Table(other_table) from u
do {
  table := update(table, u, other_table);
}

upon u =/= self
do {
  send Table(table) to u;
}

upon in(u, unblocked)
 /\ obj local object
 /\ id(obj) = o
do {
  send Object(obj) to u;
  table := register(o, u, table, 1);
  delete object obj;
}

upon receive Object(obj) from u
 /\ id(obj) = o
do {
  table := register(o, self, table, 0);
}

upon receive Message(o, p) from u
 /\ obj local object
 /\ id(obj) = o
do {
  deliver Message(o, p);
}

upon receive Message(o, p) from u1
 /\ next(o, table, union(unblocked, singleton(self))) = u2
 /\ u2 =/= self
do {
  send Message(o, p) to u2;
}
```

```
upon dispatch Message(o, p)
 /\ next(o, table, union(unblocked, singleton(self))) = u
do {
  send Message(o, p) to u;
}

upon receive Message(o, p) from u
 /\ next(o, table, union(unblocked, singleton(self))) undefined
do {
  send Message(o, p) to self;
}

upon dispatch Message(o, p)
 /\ next(o, table, union(unblocked, singleton(self))) undefined
do {
  send Message(o, p) to self;
}
```

Appendix D

Shutdown Protocol Transition System Model

D.1 Runtime Configurations

A runtime configuration is a collection *net* of nodes and arcs, as defined in Table D.1. A *valid* configuration is either the empty configuration ϵ or a configuration that can be reached by applying the reduction rules, given below, starting from a valid configuration. Note that in valid configurations, nodes have a self-loop arc, and there are always arcs in both directions between two nodes, if at all.

u		node identifier
U		set of node identifiers
q		queue of *msg*
net	$::=$	$\epsilon \mid net\,net' \mid node \mid arc$
arc	$::=$	$\mathsf{ar}\,(u, q, u')$
$node$	$::=$	$\mathsf{nd}\,(u, U_{unblk}, U_{blk}, st, U_{rdy}, U_{prep}, U_{abrt}, U_{shtdn}, U_{ack})$
st	$::=$	$\mathsf{IDLE} \mid \mathsf{TRANSACT} \mid \mathsf{ABORT} \mid \mathsf{CLEAR} \mid \mathsf{SHUTDOWN}$
msg	$::=$	NOTIFY $\mid$ PREPARE $\mid$ READY $\mid$ ABORT $\mid$ SHUTDOWN $\mid$ ACK

Table D.1: Transition system runtime configuration syntax

D.2 Reduction Rules

The reduction rules are defined in the rewriting logic style [40]. The use of curly brackets around a configuration in a rule indicates that the reduction takes place at the global level, as opposed to rules without brackets, that change only part of a configuration. Table D.2 lists the predicates and functions used in the rules. The rules sometimes use $\overline{U}_n$ for U_{unblk}, U_{blk}, and $\overline{U}_m$ for U_{rdy}, U_{prep}, U_{abrt}, U_{shtdn}, U_{ack}, or sublists, as determined by context. Figures D.1, D.2, and D.3 show the rules.

$\texttt{fresh}\,(u, \mathit{net})$	for all u' that are identifiers of nodes in net, $u' < u$
$\texttt{first}\,(q)$	the first (top) message msg in the queue q, if any
$\texttt{dequeue}\,(q)$	q with the first (top) message removed, if q is nonempty
$\texttt{enqueue}\,(q, \mathit{msg})$	q with msg added to the end
$\texttt{add\_arcs}(u, \mathit{net})$	net with arcs (with empty queues) added, connecting nodes in net and u, in both directions
$\texttt{rm\_arcs\_blk}(u, \mathit{net})$	net with all arcs to or from u removed, and u removed from the set U_{blk} in all nodes in net

Table D.2: Transition system predicates and functions

$$\frac{\begin{array}{c}(\textsc{Init})\\ \texttt{fresh}\,(u, \mathit{net}\;\mathit{net}')\end{array}}{\{\mathit{net}\;\mathit{net}'\} \to \{\mathsf{nd}\,(u, \emptyset, \emptyset, \mathsf{IDLE}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)\;\mathsf{ar}\,(u, \mathsf{emp}, u)\;\texttt{add\_arcs}(u, \mathit{net})\;\mathit{net}'\}}$$

$$\frac{\begin{array}{c}(\textsc{Shutdown})\\ U_{unblk} \cup U_{blk} = U_{ack}\end{array}}{\{\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{SHUTDOWN}, U_{ack}, \overline{U}_m)\;\mathit{net}\} \to \{\texttt{rm\_arcs\_blk}(u, \mathit{net})\}}$$

$$\frac{\begin{array}{c}(\textsc{Shutdown-singleton})\\ U_{unblk} \cup U_{blk} = \emptyset\end{array}}{\{\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\;\mathsf{ar}\,(u, \mathsf{emp}, u)\;\mathit{net}\} \to \{\texttt{rm\_arcs\_blk}(u, \mathit{net})\}}$$

$$\frac{\begin{array}{l}(\textsc{Recv-notify-unknown})\\ \texttt{first}\,(q) = \textsc{Notify} \quad \texttt{dequeue}\,(q) = q'' \quad u' \notin U_{unblk} \cup U_{blk}\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\;\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\;\mathsf{ar}\,(u, q', u') \to\\ \mathsf{ar}\,(u', q'', u)\;\mathsf{nd}\,(u, U_{unblk} \cup \{u'\}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\;\mathsf{ar}\,(u, \texttt{enqueue}\,(q', \textsc{Notify}), u')\end{array}}$$

$$\frac{\begin{array}{l}(\textsc{Recv-notify-blocked})\\ \texttt{first}\,(q) = \textsc{Notify} \quad \texttt{dequeue}\,(q) = q' \quad u' \in U_{blk}\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\;\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m) \to\\ \mathsf{ar}\,(u', q', u)\;\mathsf{nd}\,(u, U_{unblk} \cup \{u'\}, U_{blk} \setminus \{u'\}, \mathsf{IDLE}, \overline{U}_m)\end{array}}$$

$$\frac{\begin{array}{l}(\textsc{Idle-recv-prepare})\\ \texttt{first}\,(q) = \textsc{Prepare} \quad \texttt{dequeue}\,(q) = q''\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\;\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\;\mathsf{ar}\,(u, q', u') \to\\ \mathsf{ar}\,(u', q'', u)\;\mathsf{nd}\,(u, U_{unblk} \setminus \{u'\}, U_{blk} \cup \{u'\}, \mathsf{IDLE}, \overline{U}_m)\;\mathsf{ar}\,(u, \texttt{enqueue}\,(q', \textsc{Ready}), u')\end{array}}$$

$$\frac{\begin{array}{l}(\textsc{Idle-recv-abort})\\ \texttt{first}\,(q) = \textsc{Abort} \quad \texttt{dequeue}\,(q) = q'\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\;\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m) \to\\ \mathsf{ar}\,(u', q', u)\;\mathsf{nd}\,(u, U_{unblk} \cup \{u'\}, U_{blk} \setminus \{u'\}, \mathsf{IDLE}, \overline{U}_m)\end{array}}$$

Figure D.1: Shutdown protocol reduction rules, part 1

$$\frac{\begin{array}{c}(\textsc{Idle-recv-shutdown})\\ \texttt{first}\,(q) = \textsc{Shutdown} \quad \texttt{dequeue}\,(q) = q''\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\,\mathsf{ar}\,(u, q', u') \rightarrow\\ \mathsf{ar}\,(u', q'', u)\,\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\,\mathsf{ar}\,(u, \texttt{enqueue}\,(q', \textsc{Ack}), u')\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Idle-send-notify})\\ u' \neq u \quad u' \notin U_{unblk} \cup U_{blk}\end{array}}{\begin{array}{c}\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m)\,\mathsf{ar}\,(u, q, u') \rightarrow\\ \mathsf{nd}\,(u, U_{unblk}, U_{blk} \cup \{u'\}, \mathsf{IDLE}, \overline{U}_m)\,\mathsf{ar}\,(u, \texttt{enqueue}\,(q, \textsc{Notify}), u')\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Idle-enter-transact})\\ U_{unblk} \neq \emptyset \quad U_{blk} = \emptyset\end{array}}{\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \overline{U}_m) \rightarrow \mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{TRANSACT}, \overline{U}_m)}$$

$$\frac{\begin{array}{c}(\textsc{Transact-send-prepare})\\ u' \in U_{unblk} \cup U_{blk} \quad u' \notin U_{prep}\end{array}}{\begin{array}{c}\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{TRANSACT}, U_{prep}, \overline{U}_m)\,\mathsf{ar}\,(u, q, u') \rightarrow\\ \mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{TRANSACT}, U_{prep} \cup \{u'\}, \overline{U}_m)\,\mathsf{ar}\,(u, \texttt{enqueue}\,(q, \textsc{Prepare}), u')\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Transact-recv-ready})\\ \texttt{first}\,(q) = \textsc{Ready} \quad \texttt{dequeue}\,(q) = q'\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{rdy}, \overline{U}_m) \rightarrow\\ \mathsf{ar}\,(u', q', u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{rdy} \cup \{u'\}, \overline{U}_m)\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Transact-recv-prepare-lt})\\ \texttt{first}\,(q) = \textsc{Prepare} \quad \texttt{dequeue}\,(q) = q'' \quad u' < u\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{TRANSACT}, \overline{U}_m)\,\mathsf{ar}\,(u, q', u') \rightarrow\\ \mathsf{ar}\,(u', q'', u)\,\mathsf{nd}\,(u, U_{unblk} \setminus \{u'\}, U_{blk} \cup \{u'\}, \mathsf{ABORT}, \overline{U}_m)\,\mathsf{ar}\,(u, \texttt{enqueue}\,(q', \textsc{Ready}), u')\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Transact-enter-clear})\\ U_{rdy} = U_{unblk} \cup U_{blk}\end{array}}{\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{TRANSACT}, U_{rdy}, \overline{U}_m) \rightarrow \mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{CLEAR}, U_{rdy}, \overline{U}_m)}$$

$$\frac{\begin{array}{l}(\textsc{Transact-recv-prepare-nlt-in-prep})\\ \texttt{first}\,(q) = \textsc{Prepare} \quad \neg\, u' < u\\ u' \in U_{prep} \quad \texttt{dequeue}\,(q) = q'\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{prep}, \overline{U}_m) \rightarrow\\ \mathsf{ar}\,(u', q', u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{prep}, \overline{U}_m)\end{array}}$$

$$\frac{\begin{array}{l}(\textsc{Transact-recv-prepare-nlt-notin-prep})\\ \texttt{first}\,(q) = \textsc{Prepare} \quad \texttt{dequeue}\,(q) = q'' \quad \neg\, u' < u \quad u' \notin U_{prep}\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{prep}, \overline{U}_m)\,\mathsf{ar}\,(u, q', u') \rightarrow\\ \mathsf{ar}\,(u', q'', u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{TRANSACT}, U_{prep} \cup \{u'\}, \overline{U}_m)\,\mathsf{ar}\,(u, \texttt{enqueue}\,(q', \textsc{Prepare}), u')\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Abort-recv-ready})\\ \texttt{first}\,(q) = \textsc{Ready} \quad \texttt{dequeue}\,(q) = q'\end{array}}{\mathsf{ar}\,(u', q, u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{ABORT}, U_{rdy}, \overline{U}_m) \rightarrow \mathsf{ar}\,(u', q', u)\,\mathsf{nd}\,(u, \overline{U}_n, \mathsf{ABORT}, U_{rdy} \cup \{u'\}, \overline{U}_m)}$$

Figure D.2: Shutdown protocol reduction rules, part 2

$$\frac{\begin{array}{l}(\textsc{Abort-recv-prepare-have-sent-lt})\\ \mathtt{first}\,(q) = \textsc{Prepare} \quad \mathtt{dequeue}\,(q) = q'' \quad u' \in U_{prep} \quad u' \notin U_{rdy} \quad u' < u\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\ \mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{ABORT}, U_{rdy}, U_{prep}, \overline{U}_m)\ \mathsf{ar}\,(u, q', u') \rightarrow\\ \mathsf{ar}\,(u', q'', u)\ \mathsf{nd}\,(u, U_{unblk} \setminus \{u'\}, U_{blk} \cup \{u'\}, \mathsf{ABORT}, U_{rdy}, U_{prep}, \overline{U}_m)\\ \mathsf{ar}\,(u, \mathtt{enqueue}\,(q', \textsc{Ready}), u')\end{array}}$$

$$\frac{\begin{array}{l}(\textsc{Abort-recv-prepare-have-sent-nlt})\\ \mathtt{first}\,(q) = \textsc{Prepare} \quad \mathtt{dequeue}\,(q) = q'\\ u' \in U_{prep} \quad u' \notin U_{rdy} \quad \neg\, u' < u\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\ \mathsf{nd}\,(u, \overline{U}_n, \mathsf{ABORT}, U_{rdy}, U_{prep}, \overline{U}_m) \rightarrow\\ \mathsf{ar}\,(u', q', u)\ \mathsf{nd}\,(u, \overline{U}_n, \mathsf{ABORT}, U_{rdy}, U_{prep}, \overline{U}_m)\end{array}}$$

$$\frac{\begin{array}{l}(\textsc{Abort-send-abort})\\ u' \in U_{unblk} \quad u' \in U_{rdy} \quad u' \notin U_{abrt}\end{array}}{\begin{array}{c}\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{ABORT}, U_{rdy}, U_{abrt}, \overline{U}_m)\ \mathsf{ar}\,(u, q, u') \rightarrow\\ \mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{ABORT}, U_{rdy}, U_{abrt} \cup \{u'\}, \overline{U}_m)\ \mathsf{ar}\,(u, \mathtt{enqueue}\,(q, \textsc{Abort}), u')\end{array}}$$

$$\frac{\begin{array}{l}(\textsc{Abort-enter-idle})\\ U_{unblk} \cap U_{prep} = U_{abrt}\end{array}}{\begin{array}{c}\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{ABORT}, U_{rdy}, U_{prep}, U_{abrt}, U_{shtdn}, U_{ack}) \rightarrow\\ \mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{IDLE}, \emptyset, \emptyset, \emptyset, U_{shtdn}, U_{ack})\end{array}}$$

$$\frac{(\textsc{Clear-enter-shutdown})}{\mathsf{nd}\,(u, \overline{U}_n, \mathsf{CLEAR}, \overline{U}_m)\ \mathsf{ar}\,(u, \mathsf{emp}, u) \rightarrow \mathsf{nd}\,(u, \overline{U}_n, \mathsf{SHUTDOWN}, \overline{U}_m)\ \mathsf{ar}\,(u, \mathsf{emp}, u)}$$

$$\frac{\begin{array}{l}(\textsc{Shutdown-recv-ack})\\ \mathtt{first}\,(q) = \textsc{Ack} \quad \mathtt{dequeue}\,(q) = q'\end{array}}{\begin{array}{c}\mathsf{ar}\,(u', q, u)\ \mathsf{nd}\,(u, \overline{U}_n, \mathsf{SHUTDOWN}, U_{ack}, \overline{U}_m) \rightarrow\\ \mathsf{ar}\,(u', q', u)\ \mathsf{nd}\,(u, \overline{U}_n, \mathsf{SHUTDOWN}, U_{ack} \cup \{u'\}, \overline{U}_m)\end{array}}$$

$$\frac{\begin{array}{l}(\textsc{Shutdown-send-shutdown})\\ u' \in U_{unblk} \cup U_{blk} \quad u' \notin U_{shtdn}\end{array}}{\begin{array}{c}\mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{SHUTDOWN}, U_{shtdn}, \overline{U}_m)\ \mathsf{ar}\,(u, q, u') \rightarrow\\ \mathsf{nd}\,(u, U_{unblk}, U_{blk}, \mathsf{SHUTDOWN}, U_{shtdn} \cup \{u'\}, \overline{U}_m)\ \mathsf{ar}\,(u, \mathtt{enqueue}\,(q, \textsc{Shutdown}), u')\end{array}}$$

Figure D.3: Shutdown protocol reduction rules, part 3