Concurrent Architecture for a Multi-agent Platform

Michael Duvigneau, Daniel Moldt, Heiko Rölke

Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, 22527 Hamburg, Germany
5duvigne,moldt,roelke@informatik.uni-hamburg.de

Abstract. A multi-agent system has a high degree of concurrency. Petri nets are a well-established means for the description of concurrent systems. Reference nets are higher level, object-oriented Petri nets. With Renew (REference NEt Workshop), there exists a tool to model and execute reference nets with seamless Java integration. So, reference nets can be used to design executable multi-agent systems while hiding the sometimes annoying details of concurrent implementations in traditional programming languages. The technique is currently used to implement a FIPA-compliant agent platform for multi-agent systems (called CAPA) focused on retaining a maximum level of concurrency in the system.

1 Introduction

Multi-agent systems implicate a high degree of concurrency: Agents operate independently from each other and can engage themselves in several tasks simultaneously. But most conventional programming languages and therefore the agent frameworks built upon them have only restricted support for concurrent systems. A lot of syntactical or management overhead is needed when implementing concurrent systems using such techniques, which blurs the view on the essential concurrency and synchronisation concepts. So, the sequential view of conventional programming languages leads to systems which do not provide maximum concurrency – it would be better to use a technique where concurrency is the basic assumption, and where it can be explicitly restricted when inappropriate.

Petri nets provide a graphical-intuitive model with formal and precise semantics to handle concurrency and synchronisation. With the extensions of higher level nets, e.g. colored Petri nets, object oriented nets or reference nets, nets can be used to model multi-agent systems efficiently. In our approach we mainly use reference nets [11] because of their object-oriented character, their support of the "nets within nets" paradigm, and the availability of the tool Renew (Reference Net Workshop, [10]) which is able to execute reference nets with seamless Java integration in its simulation engine.

The Mulan (Multi Agent Nets) architecture presented in [14] uses reference nets to describe four levels of multi-agent systems from the overall system view down to the agent-behavior modeling protocols. Although the Mulan model can

Giunchiglia, Odell, Weiß: Agent-Oriented Software Engineering III, LNCS 2585, pp. 59–72, 2003.
© Springer-Verlag Berlin Heidelberg 2003

be executed using the Renew tool, some practical features needed for interoperability with other agent platforms are missing. This is due to the conceptual character of Mulan, where the inter-platform communication structures are modeled as a net, allowing cross-platform communication between agents within the net simulation only. To enable the agents to communicate with agents on other platforms, either located at another host or implemented differently (or both), the conceptual platform net of Mulan needs to be extended by a platform implementation oriented along the specifications generated by FIPA (Foundation for Intelligent Physical Agents, [4]). The architecture of this agent platform, called CAPA (Concurrent Agent Platform Architecture), is the subject of this paper.

2 Reference Nets and Concurrency

As the basic technique for modeling agent systems we use reference nets which are a special kind of high-level Petri nets.[1] In this contribution it is assumed that the reader has some knowledge about Petri nets in general. However, the relevant features of Petri nets and reference nets for this paper will be sketched in the following. These are folding, concurrency, and some net extensions.

The basic net formalism of C/E-nets (Condition/Event-nets) comprises all basic features of nets: sequence, synchronisation, conflict, and concurrency. These concepts are also present in more abstract definitions like P/T-nets (Place/Transition-nets) or reference nets. Concurrency can be found in Fig. 1, where the transitions `a` and `b` can fire independently, simultaneous firing included.

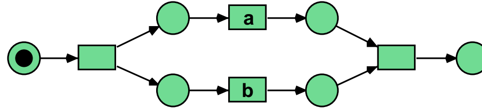

Fig. 1. Concurrent transitions in a C/E-net

In Fig. 2 the concept of folding is shown. Folding can clarify net drawings by combining similar structures of a net into one (parameterised) structure. The net in Fig. 2a can be fold in two different ways: the result in Fig. 2b drops some information and stays at the level of P/T-nets while the net in Fig. 2c preserves all information by distinguishing the two tokens by colors.

Fig. 3 shows the folded net of Fig. 1. It is important to notice that even if the structure of the resulting net in Fig. 3 seems to be sequential the behavior remains fully concurrent as the net from Fig. 1.

[1] An introduction to the capabilities of reference nets and their usage is given in [9]. The documentation shipped with Renew [10] provides a more detailed description

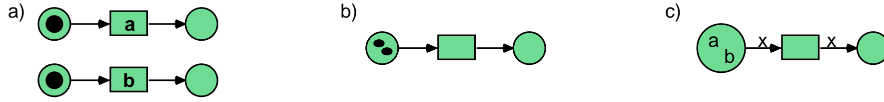

Fig. 2. Folding

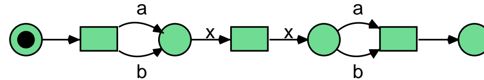

Fig. 3. The net of Fig. 1 folded as a colored net

In our approach agents are represented by net instances as a specific feature of reference nets. The static structure of a drawn net can be considered to be a class while net instances are objects of the type of the related static net model. Each instance has its own local marking, representing the state of the object. Several instances therefore introduce concurrent behavior if they have at least one activated transition each. Even more concurrency can be found if there are (folded) concurrent parts in the net, which is the usual case for us. The instances can be agents.

Reference nets allow communication between the instances by synchronous channels. In Fig. 4 the main concepts can be seen. The transition in the system net can use the reference ref to synchronise itself with transition b of the object net through the synchronous channel :lookup. Both transitions have to fire synchronously – if either one is not activated, the other cannot fire, too. The information flow through the channel is bidirectional, so that transition b in this example can bind the second channel parameter based on the first parameter bound by transition a, resulting in the variable y bound to "6x7". Synchronous channels are the means for agents to communicate with their environment and other agents.

The IDE (Integrated Development Environment) used for our approach is Renew in combination with the Mulan architecture. This allows to build models and systems at the same time, since reference nets are directly executable within the Renew simulation engine. Based on the agent concepts each net instance can be replaced by or connected to Java objects.

The new agent platform, called CAPA (Concurrent Agent Platform Architecture), is designed and implemented under the guideline of keeping the level of concurrency as high as possible. The seamless Java integration of the Renew tool

of syntax and features of reference nets. The full reference net formalism, including its theoretical foundations, is explained in [11].

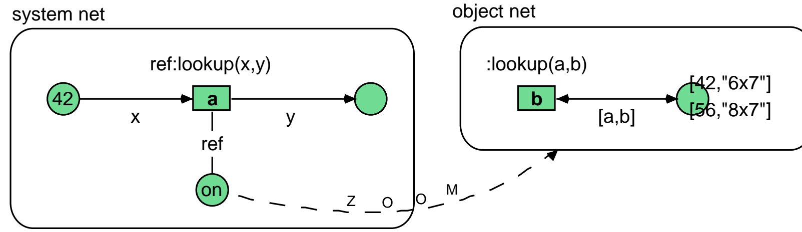

Fig. 4. Net instances communicating through a synchronous channel

allows to implement the agent platform in a mixture of Java and reference nets. Therefore, the advantage of reference nets in handling concurrency and synchronisation can be combined with the flexibility of the object-oriented programming language when working with abstract data types or using the functionality provided by Java's huge class library.

3 Multi-Agent Nets

The multi-agent system architecture Mulan [8] is based on the "nets within nets" paradigm [17], which is used to describe the natural hierarchies in an agent system. Mulan is implemented in Renew. Mulan has the general structure as depicted in figure 5[2]: Each box describes one level of abstraction in terms of a system net. Each system net contains object nets, which structure is made visible by the ZOOM lines.[3]

The net in the upper left describes an agent system, whose places contain agent platforms as tokens. The transitions describe communication or mobility channels, which build up the infrastructure. This is just an illustrating example, the number of places and transitions or their interconnection has no further meaning.

By zooming into the platform token on place p1, the structure of a platform becomes visible, shown in the upper right box. The central place agents hosts all agents, which are currently on this platform. Each platform offers services to the agents, some of which are indicated in the figure.[4] Agents can be created (transition new) or destroyed (transition destroy). Agents can communicate by message exchange. Two agents on the same platform can communicate by the transition internal communication, which binds two agents, the sender and the

[2] This is just a simplified version, since for example only some nodes are shown and all synchronous channels are omitted.

[3] This zooming into net tokens should not to be confused with place refining.

[4] Note that only mandatory services are mentioned here. A typical platform will offer more and specialised services, for example implemented by special service agents.

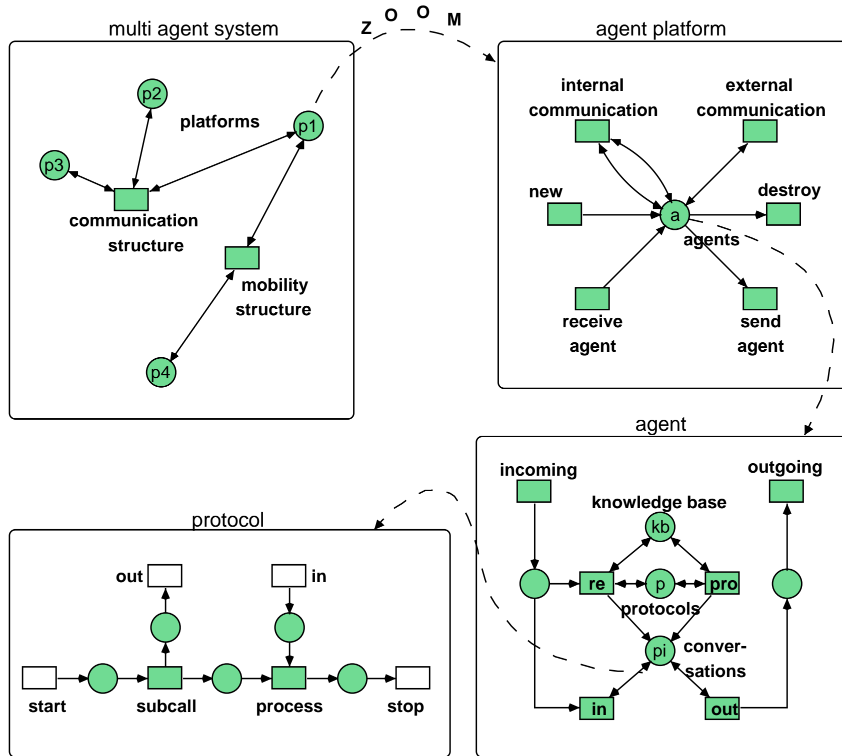

Fig. 5. Agent systems as nets within nets

receiver, to pass one message over a synchronous channel.[5] External communication (external communication) only binds one agent, since the other agent is bound on a second platform somewhere else in the agent system. Also mobility facilities are provided on a platform: agents can leave the platform via the transition send agent or transitions could enter the platform via the transition receive agent from another platform.

Agents are also modeled in terms of nets. They are encapsulated, since the only way of interaction is by message passing. Agents can be intelligent, since they have access to a knowledge base. The behavior of the agent is described in terms of protocols, which are again nets. Protocols are located as templates on the place protocols. Protocol templates can be instantiated, which happens for example if a message arrives. An instantiated protocol is part of a conversation

[5] This is just a technical point, since via synchronous channels provided by Renew asynchronous message exchange is implemented.

and lies in the place conversations. The detailed structure of protocols and their interaction have been addressed before in [8], so we skip the details here.

4 Concurrent FIPA-compliant Multi-agent Platform

Mulan is extended by a FIPA-compliant agent platform, called CAPA (Concurrent Agent Platform Architecture), in order to allow cross-platform communications. The new platform replaces the conceptual platform net described in the previous section. The implementation tries to keep the highest level of concurrency by taking advantage from the possibility of mixing reference nets with Java code.

To comply with the FIPA-2000 set of specifications, CAPA has to provide for its agents the management and directory services AMS (Agent Management System) and DF (Directory Facilitator), a local Message Transport System (MTS), and an interface for communication with external platforms, the Agent Communication Channel (ACC). FIPA-agents communicate using asynchronous messages expressed in the FIPA ACL (Agent Communication Language), so an internal representation for such messages is useful to simplify the message interpretation of the agents.

4.1 Message Representation

ACL Messages are represented internally by objects following the key-value-tuple concept given in the “FIPA Abstract Architecture” [4]. Many other information structures used in messages, for example of the FIPA agent management ontology, are represented using the same key-value-tuple concept. In combination with an similar value-tuple concept (without keys), these representation classes can be used for several content languages and ontologies.[6]

A subsumption relation is defined upon both tuple classes to allow the agent developer to use pattern matching in a similar way he can use the unification mechanism included in reference nets. The possibility to use flexible pattern matching saves the agent developer a lot of work when it comes to the analysis of incoming messages.

For the implementation of the tuple classes there are two alternatives: Both techniques, Java code or reference nets, could be used. The reference net implementation of a key-value-tuple would be easy, since the formalism allows to store the key-value-pairs as tuples in places. The pairs could be retrieved by using unification for pattern matching with the key component of the tuples. The Java implementation could build up upon existing classes implementing the `java.util.Map` or `List` interfaces.

The main disadvantage of the net implementation is due to the lack of type-checking and inheritance when using reference nets with synchronous channels

[6] The reduction of different languages and ontologies to a generic tuple concept has been inspired by the JAS architecture (see [6]).

as interface: the net implementation would not be able to catch many of the small careless mistakes made by a developer while he is sketching his agents.

But the decision for a Java implementation has its drawbacks, too. A mutable, `Map`-based implementation of a container class has to be protected against concurrent modifications. The `synchronise` feature of the Java language, which could solve some of the concurrency-related issues, does not combine well with the synchronisation scheme of Petri nets, because its text-based locking scheme cannot be used to lock an object's monitor across several transitions.

Nevertheless, CAPA message representation is currently done in Java, for the advantage of type checking and inheritance, allowing for convenience methods that simplify the agent developer's work. The synchronisation problem is delegated to the agent implementation – it has to avoid access conflicts. An alternative solution could be the use of immutable representation objects where instead of every modification a new object gets instantiated.

4.2 Internal Interface to the Message Transport System

The interface between agents and the internal message transport system consists of two synchronous channels (as depicted in Fig. 6). The agent has to provide two uplinks, namely :receive(message) and :send(message), to which the transport system connects via appropriate downlinks. The downlinks of the system net need a reference to be established, this can be obtained through a test arc from the active agents place. While the :send channel can be activated for any message/agent combination, the :receive channel should only be triggered if the message's receiver entry matches the receiver reference used to establish the channel.

The transport system interface could be specified alternatively by declaring two Java methods. The transport system has to provide `send(message)` while the agent offers a `receive(message)` method. But there is one main difference between the synchronous channel interface and the Java-style interface: The bidirectional information flow through a synchronous channel allows the same reference direction to be used for the receive as well as for the send channel. The object or net instance modeling the agent does not need a reference to the transport system to send its messages. Instead, the environment of the agent connects to the agent by using its reference from the platform management. Despite of the "backward" direction of this connection, the agent can trigger the send channel to the transport system at any time by putting a message in its outgoing message place.[7]

The advantage of having an asymmetric relation between the agent platform and agents is a simple security aspect: Since the agent does not have a reference to the transport system or any other component of the platform implementation, it cannot abuse these references. Further, the platform can easily install a

[7] Here it is assumed, of course, that the platform acts as cooperative environment for the agent and does not block the channel – but uncooperative behavior would be possible for the platform regardless of the reference direction.

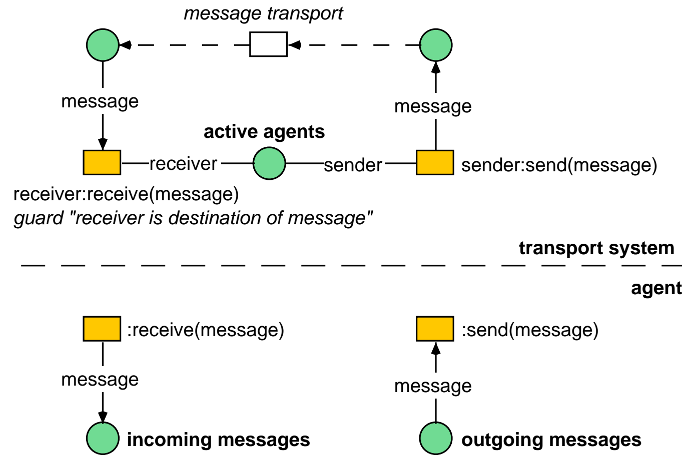

Fig. 6. The internal message transport interface

guarding instance between the agent and the transport system (without a need to reconfigure the agent), which checks all inbound and outbound transmissions for permission.

A look at the concurrency aspect of the synchronous channel interface shows that it is not restricted in any way. A synchronous channel can be established between any pair of activated transitions at any time, including the possibility of synchronising one pair of transitions several times at once, if there is a sufficient number of input tokens (e.g. messages) available. So any agent can send several messages at one time as well as several agents can send their messages simultaneously (the same holds, of course, for message delivery) – unless one of the involved nets restricts the concurrency. On the transport system side of the interface, the implementation takes care not to restrict the concurrency (for example by using test arcs to fetch the agent references). If an agent implementation wants to reduce the concurrency, it can do so without affecting other agents.

4.3 Message Transport System

The Mulan platform net distinguishes between internal (between agents on the same platform) and external (cross-platform) communication (see Fig. 5). Currently, CAPA's internal message transport system does not make this distinction (this behavior may be subject to changes). So, all messages – wherever they come from – are passed to the central `MessageTransportService` which provides the functionality of the ACC from the "FIPA Transport Service Specification" [4].

The transport system architecture is defined by two Java interfaces, one of which is the already mentioned `MessageTransportService`, the other de-

clares the functionality of individual `Transport` protocol implementations. The `MessageTransportService` inspects the message's envelope, determines all possible `Transport`s based on the envelope's destination addresses and tries the `Transport`s until one of them succeeds in forwarding the message. The internal transport net mentioned before is hooked into the transport system as one `Transport` offering the message transfer to all local agent addresses.

The default implementation of the `MessageTransportService` interface is provided by the net depicted in figure 7. This figure shows executable "source code" taken from the current implementation. The channel uplinks :transportMessage, :getDescription, :addTransport and :removeTransport act as method bodies for the Java method declarations from the `MessageTransportService` interface. To avoid cluttering the graphical representation with large code blocks, some functionality is moved into a Java class called `ACCHelper`. The methods of this class are all static and – with the exception of `tryTransport` which possibly forwards the message – free from side-effects.

The main part of the drawing looks like a sequence of loops. This rather sequential than concurrent impression is in fact correct with respect to the handling of one single message. The steps of extracting an address from the message envelope, determining a `Transport` capable of reaching that address and letting the `Transport` try to forward the message have to be done in this specific order. As the FIPA Transport Service Specification suggests, multiple addresses from a message envelope also have to be tried in the sequence given in the envelope to respect the agents' preferences. And the goal of not duplicating a message unnecessarily enforces the sequential usage of multiple `Transport`s which could be able to forward the message.

But the inherent concurrency of Petri nets comes to effect immediately if more than one message is in the system. Since all places not storing the message tuple itself (e.g. all side conditions) are connected to transitions by using test arcs, full concurrency is available. This even holds for the split parts of multicast messages: After the transition labeled Message splitting has produced several [envelope, message]-pairs (one for each addressee) from the original envelope through a flexible output arc (with two arrow tips)[8], these pairs are handled completely independent from (and concurrent to) each other.

The independency of different message handling "threads" could be represented by creating one individual instance of a (nearly unmodified) message transport net for every message. Such an implementation would come rather close to the Java concurrency concept where an instance of `Thread` must be used to handle each message in order to reach the same level of concurrency. But in Renew, the folding of several net instances into one net instance is possible without a reduction in concurrency and avoids many net instantiations. The flexible output arc used for message splitting helps in this mission because it allows the folding of the creation of a dynamical number of net instances.

[8] Flexible arcs are based upon the ideas used by Reisig in [15]. They transfer a dynamic amount of tokens, determined as a function of other input tokens. The exact semantics for reference nets are described in the manual shipped with Renew [10].

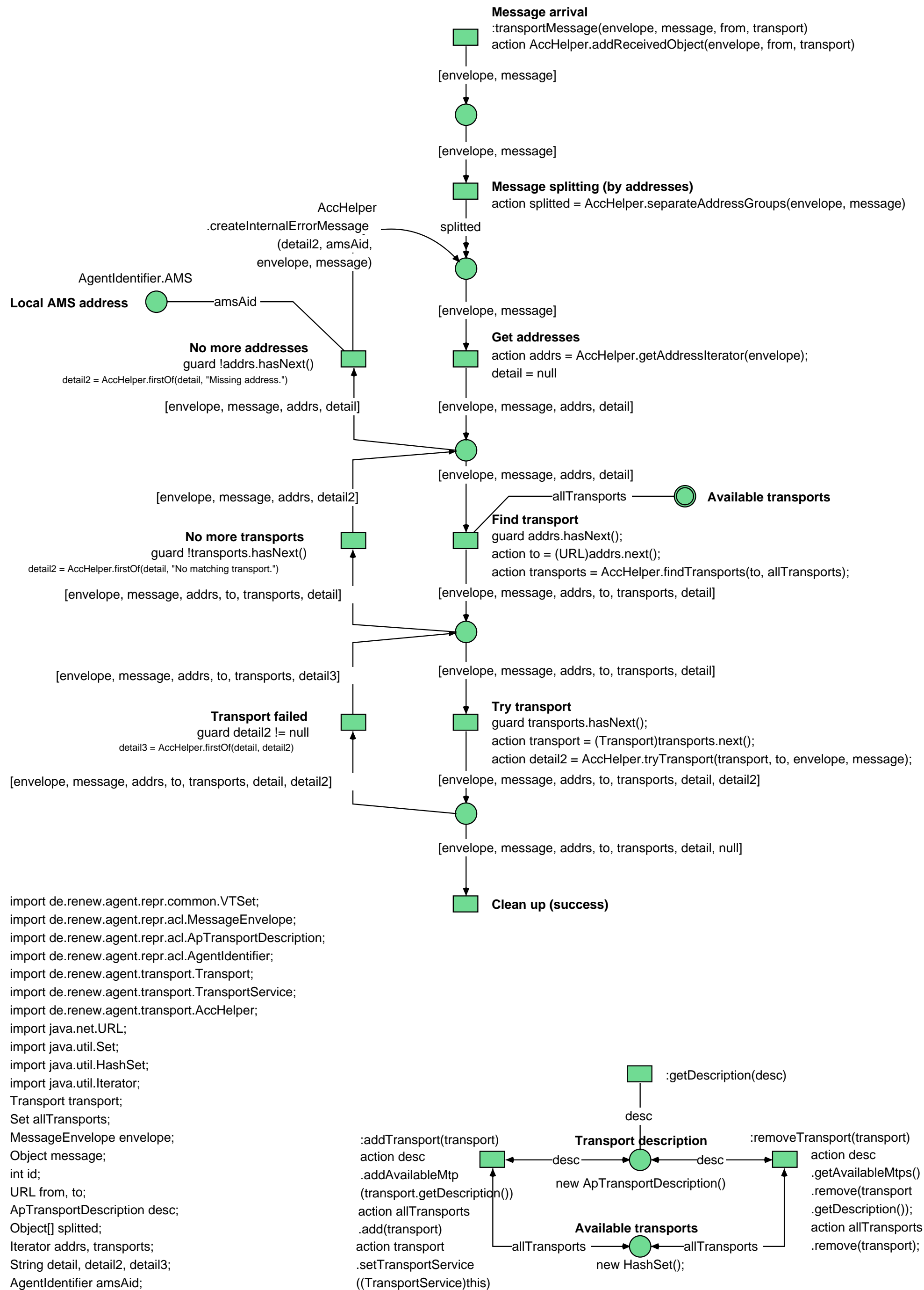

Fig. 7. `MessageTransportService` implementation

4.4 Management and Directory Services

The Agent Management System (AMS) and the Directory Facilitator (DF) are implemented as pure Mulan agents and run in CAPA like any other application-specific agent. For each of the service functions required by the FIPA Agent Management Specification there exists a protocol net which gets instantiated when a message requesting this function is delivered to the agent. The database of agent descriptions managed by AMS and DF is stored in their knowledge bases and updated by their reactive protocols.

The implementation of both agents relies heavily on the refined default implementation of the Mulan agent concept. This implementation, which serves mainly as a proof of concept with practical use, can easily be replaced by any other implementation conforming to the internal message transport interface. The current implementation consists of three basic nets:

- An agent net implements the message transport interface described before. Further it provides the glue between the knowledge base, the protocol factory and the application-specific protocols. Incoming messages which belong to running conversations are directly forwarded to the protocol instance responsible for the conversation. All other incoming messages are handled by the protocol factory (see below).
- A knowledge base net provides basic, key-value-tuple-like knowledge management. This net allows the agent's protocols to create, read and modify values for given keys. While concurrent read access is allowed, a modification of any key currently requires exclusive access to the whole database. However, the granularity of exclusive access can be changed by providing a different implementation, for example based upon a database engine.
- A protocol factory net chooses protocols to instantiate based upon incoming messages. The subsumption relation defined on the internal message representation objects enables the factory to choose a protocol in accordance to the most specific matching message pattern – allowing the agent developer to specify fall-back protocols associated to a general message pattern. The instantiation of reactive protocols can occur concurrently – as often as incoming messages are available.
  Pro-active protocol instantiation is handled by the protocol factory, too. But since a pro-active transition without preconditions can fire any number of times (even concurrent to itself), the pro-active protocol instantiation is currently restricted to a one-time-shot for practical reasons.

Based upon these three nets, any number of application-specific protocols can run simultaneously. Synchronisation between running protocols appears indirectly, when knowledge base modifications occur.

In the case of the AMS's and DF's directory functions, the search protocol can run any number of times concurrently, because it requires read access only. The other protocols modify the knowledge base by adding, removing or changing entries in the directory. Therefore, all instances of these protocols contain – along with some other preliminary transitions – one transition which requires exclusive knowledge base access (excluding all read-only protocols, too).

5 Related work

In [8], the Mulan approach has been compared with several other Petri net based agent models, like those of Sibertin-Blanc et.al. [2], Fernandes and Belo [3], Miyamoto and Kumagai [12], or Xu and Shatz [18]. The graphical models of UML [16] and the agent-oriented extensions proposed by AUML [13] do not provide all aspects covered by the reference net/Mulan approach in one diagram type: mainly the exact operational semantics are missing.

Other FIPA-compliant agent platform implementations and agent development environments exist, like FIPA-OS [5] or JADE [7]. CAPA implements again technical features of those platforms that have to be implemented by each FIPA-compliant agent platform, like message representation or transport protocols. The main difference to those platforms is that CAPA does not need to worry about task scheduling, threads or other means to provide concurrency to agents – due to the existing Renew/Mulan-environment.

The effort of JAS [6] to create a Java interface framework for FIPA-compliant agent platforms would be interesting to adopt by CAPA. Unfortunately, the JAS effort was not grown enough when the main parts of CAPA were written to integrate it from the beginning. However, there are some conceptual differences between CAPA and JAS in how agents access platform services.

6 Conclusion

The Mulan architecture extended by the CAPA platform forms an agent framework that provides concurrency at all architectural levels throughout the whole system. A software engineer designing a multi-agent system based on this framework can use as much of the concurrency as desired. The engineer gains freedom in modeling the important concurrency aspect of multi-agent systems explicitly.

And the Mulan/CAPA framework is suitable for practical use. The platform has reasonable performance for our test scenarios and is able to host agents relying on the FIPA-proposed communication structure. This has been proved by the implementation of a popular board game as a multi-agent system based on the framework in a student project at the University of Hamburg.

These features are due to the approach of specifying a FIPA compliant agent platform by using higher level Petri nets, whereas the specification can serve as executable implementation with assistance of the Renew simulation engine. The same approach is available to developers doing agent-oriented software-engineering: They can use an efficient, fast and intuitive modeling technique for concurrent systems at an abstract level – and get an executable implementation in the same step.

The graphical representation of reference nets provides an intuitive means with formal background and precise semantics for modeling concurrency and synchronisation, which both are vital concepts within multi-agent systems. So concurrency aspects can be modeled and discussed explicitly during agent development, as it has been done during the development of CAPA.

The tight integration of Java into Renew allows to integrate Java-implemented parts into the multi-agent system. The element shift from reference nets to Java or from Java to reference nets leads to an abstraction mechanism that combines components from the different implementation techniques at the object or agent level. The result is a clear decomposition of the system or model, using aggregation as main relation concept.

The combination of Java and Renew as base technologies for the agent platform has a couple of other advantages. The independency from technical platforms provided by the Java runtime system allows the agent platform to run in many technical environments. Java's object-oriented type system and huge class library make the development and integration of application-specific functionality into the agent system easier.

The simulation of the running system by the Renew engine is animated and can be inspected interactively, hence allowing validation of the built models and systems. Using Petri nets for modeling multi-agent systems paves the way to use existing methods and tools for formal Petri net analysis. These tools and methods allow the developer to analyse and verify specific sub-cases of the nets which have already been drawn during the development process.

In the context of Mulan, the support of agent mobility has already been tried out, with weak and strong notions.[9] CAPA is able to support different mobility levels – a weaker mobility where the agent has to stop all activities and extract its knowledge base before it can move is possible as well as transparent serialisation of a running agent net instance with complete state transfer.

CAPA is on the way to become a FIPA-compliant agent platform. The required communication infrastructure is already available, but it is currently lacking a FIPA-compliant transport protocol. The platform is designed and implemented with the integration of such a transport protocol in mind, but the concrete implementation of the protocol has not been done yet. Therefore, the interoperability with other FIPA-compliant platforms could not be tested up to now, but will be done soon.

The protocol-driven agent model described in section 4.4 is not mandatory for the use of CAPA. As long as it offers the synchronous channels required by the internal message transport interface, any agent model – reactive or deliberative – can be implemented and inserted into the agent system.

The future plan for the Mulan, CAPA, and Renew combination is to provide a fully FIPA-compliant agent platform integrated into an IDE for the graphical development of agents and multi-agent systems.

References

1. L. Bettini and R. De Nicola: Translating Strong Mobility into Weak Mobility. In G. P. Picco, editor, Mobile Agents, volume 2240 of LNCS, p. 182 pp. Springer 2001

[9] The distinction between weak and strong mobility has been discussed in [1].

2. W. Chainbi, C. Hanachi, and C. Sibertin-Blanc: The Multi-agent Prey/Predator problem: A Petri net solution. In P. Borne, J.C. Gentina, E. Craye, and S. El Khattabi, editors, Proceedings of the Symposium on Discrete Events and Manufacturing systems, Lille, France, 1996. CESA'96 IMACS Multi-conference on Computational Engineering in System Applications.
3. J.M. Fernandes and O. Belo: Modeling Multi-Agent Systems Activities Through Colored Petri Nets. In 16th IASTED International Conference on Applied Informatics (AI'98), pp. 17–20, Garmisch-Partenkirchen, Germany, Feb. 1998.
4. Foundation for Intelligent Physical Agents (FIPA). Specifications. 2001. Represented at http://www.fipa.org.
5. FIPA Open Source (FIPA-OS). 2001. Available at http://fipa-os.sourceforge.net.
6. Java Agent Services Specification (JAS). 2001. Available at http://www.java-agent.org.
7. F. Bellifemine, G. Rimassa, A. Poggi, T. Trucco, G. Caire and F. Bergenti: Java Agent Development Framework (JADE). 2002. Available at http://sharon.cselt.it/projects/jade.
8. M. Köhler, D. Moldt, and H. Rölke: Modeling the behaviour of Petri net agents. In J. M. Colom and M. Koutny, editors, Proceedings of the 22nd Conference on Application and Theory of Petri Nets, volume 2075 of LNCS, pp. 224–241, Springer 2001.
9. O. Kummer: Introduction to Petri Nets and Reference Nets. Sozionik aktuell, No. 1, 2001. ISSN 1617-2477. Available at http://www.sozionik-aktuell.de.
10. O. Kummer, F. Wienberg and M. Duvigneau: Reference Net Workshop (Renew). Universität Hamburg 2001. Available at http://www.renew.de.
11. O. Kummer: Referenznetze. Dissertation, Universität Hamburg, 2002.
12. T. Miyamoto and S. Kumagai: A Multi Agent Net Model of Autonomous Distributed Systems. In Proceedings of CESA 96, Symposium on Discrete Events and Manufacturing Systems, pp. 619–623, 1996.
13. J. Odell, H. Van Dyke Parunak and B. Bauer: Extending UML for Agents In G. Wagner, Y. Lesperance and E. Yu, editors, Proceedings of the Agent-Oriented Information Systems (AOIS) Workshop at the 17th National conference on Artificial Intelligence (AAAI), Austin, TX, pp. 3–17, 2000.
14. H. Rölke: Mulan: Modellierung und Simulation von Agenten und Multiagentensystemen mit Referenznetzen. Technical report. Universität Hamburg, Fachbereich Informatik 2002.
15. W. Reisig: Elements of Distributed Algorithms. Springer, Berlin 1998.
16. Unified Modeling Language (UML). Object Management Group (OMG) 2001, Available at http://www.omg.org.
17. R. Valk: Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, Application and Theory of Petri Nets, volume 1420 of LNCS, pp. 1–25. Springer 1998.
18. H. Xu and S.M. Shatz: A Framework for Modeling Agent-Oriented Software. In Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, April 2001.