Component-based Robotic Engineering Part II: Systems and Models

Davide Brugali, *Member, IEEE* and Azamat Shakhimardanov

***Index Terms*—Software Engineering, Reuse, Architectures.**

I. Introduction

This article is the second of a two-part series intended as an introduction to Component-based Software Engineering (CBSE) in Robotics.

In Part I, we regarded a component as a piece of software that implements robotic functionality. The focus was on design principles and implementation guidelines that enable the development of reusable and maintainable software building blocks.

In Part II, we discuss the role of software components as architectural units of large, possibly distributed, software-intensive robotic systems. The focus is on technologies to manage the heterogeneity of hardware, computational, and communication resources and on design techniques to assemble components into systems.

A Component-based System is a composition of components and the way components interact with other components and with the computational environment greatly affects the flexibility of the entire system and the reusability of individual functionality.

The paper is structured as follows. Section II discusses the challenges that make the development of reusable components and flexible component systems difficult in robotics. In particular, we analyze the different sources of variability in robotic technology and applications and the corresponding requirements for flexibility in robotic software component-based systems. The subsequent four sections introduce key concepts related to the development of reusable software components, such as threading, synchronization, resource awareness, distribution, and quality of service negotiation and illustrate architectural models that enable the independent evolution of different variability concerns of a component-based robotic system. They are classified in four categories related to main concerns of component-based systems, namely Computation (Section III), Configuration (Section IV), Communication (Section V), and Coordination (Section VI). Finally, Section VII draws the relevant conclusions.

It should be noted that this paper is not a survey of the state-of-the-art in component-based software engineering, nor in robotics software development. While there is an increasing number of research projects facing the challenges of engineering the software development process in robotics, for which a survey of their recent achievements would be a valuable contribution to the literature, the goal of this paper is to present an overview of software design principles that enable the development of flexible and reusable robotic software systems. With this goal in mind, we present both a classification and a possible architectural interpretation of well-known concepts and recent advances in component-based software engineering. Where appropriate, we indicate the similarities between the architectural models presented in this paper and some of the models that have been documented in the literature.

D. Brugali is with the Department of Computer Science and Mathematics, Universita' degli Studi di Bergamo, Dalmine, Italy e-mail: davide.brugali@unibg.it.

A. Shakhimardanov is with the Department of XXX, Bonn-Rhein-Sieg University, Germany e-mail: azamat.shakhimardanov@fh-brs.de

Manuscript received xxx xx, 2009; revised xxx xx, 2009.

II. Challenges in building robotic software component systems

A number of issues make the design and development of robotic component-based systems difficult. Robotics is an experimental science that can be analyzed from a double perspective.

On one hand, it is a discipline that has its roots in mechanics, electronics, computer science and the cognitive sciences. In this regard, software components embed functionality and control laws that bring together advanced research results built on quickly-changing technologies.

On the other hand, Robotics is a research field that pursues ambitious goals, such as the study of intelligent behavior in artificial systems. These means that robots are used in ever new use cases and application scenarios with different functional requirements. Software components with similar functionality are integrated in more and more complex component-based systems with increasing demanding performance and reliability requirements.

The intrinsic change-centric nature of robotic applications, poses the challenge of designing software components and systems that are flexible enough to accommodate frequently changing functional and non-functional requirements. The IEEE Standard Glossary of Software Engineering Terminology defines flexibility as the "ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed".

The term environment refers to the complete range of elements in an installation that interact with the component-based software system. This includes the computer and network platform, the controlled robotic hardware, and the robotic applications that integrate the reusable components.

More specifically, flexibility is concerned with the portability of the component system on different computational environments (e.g. from a centralized to a distributed system), the

interoperability among independently developed components (e.g. components interfacing heterogeneous robotic devices), the reusability of individual components in different application contexts (e.g. a motion planner for static or dynamic environments), and the component system reconfigurability at runtime (e.g. adaptable robot behaviors).

The key to achieving software flexibility is the possibility to predict the class of changes that are likely to occur in the environment over the lifespan of robotic software components and that affect components and systems portability, reusability, interoperability, and reconfigurability.

A. Functional variability

Sensing, planning, control, reasoning, and learning are human-like capabilities that can be artificially replicated in a computer-based robotic system as software applications [1].

In complex robot control applications, several concurrent activities access the hardware devices (e.g. 3D perception for obstacle avoidance and for map building), control complex mechanisms (e.g. visual servoing for a mobile manipulator), or coordinate a set of subsystems (e.g. coalition formation for a team of robot). Each robot activity has its own timing requirements and the interaction between subsystems can be either synchronous or asynchronous. Typically, low-level control loops regulating the robot motion require the synchronous data flow model of computation, where control software periodically requests new measurements from sensors (e.g. encoders) and sends motion commands to the actuators. In contrast, the asynchronous event triggering model of computation simplifies the implementation of higher-level control loops for monitoring the robot environment.

Software implementations of common robot functionality (e.g. motion planning, navigation, manipulation) may substantially differ for extra-functional properties, such as performance, completeness, and resource demand. They are often tied to specific robotic platforms, because the information about the robot morphology, kinematics, and dynamics are typically scattered throughout the code of the control application and are represented using heterogeneous data structures. They also make implicit assumption about the operational environment or the robot task.

In order to enhance reusability of software implementations of common robot functionality, there is the need to make software dependencies to the robotic platform, operational environment and application context explicit. A few research projects have dealt with this issue, such as the CLARAty mechanism model [2], which pursues the reusability of control applications for different robotic mechanisms by explicitly representing their mechanical structure. Such a model should enable the description of the structural and behavioral properties of each mechanism part or subsystem, the relationships and constraints among the parts, and the topology of the system.

B. Hardware variability

Robotic platforms have onboard computing hardware, which can range from microprocessors to PLC and general purpose processors (e.g. laptop or PC), often has severe constraints on computational resources, storage, and power, and is interfaced to a multitude of highly heterogeneous sensors and actuators, such as laser range finders, stereo cameras, global positioning systems (GPS), servo motors, grippers, wheeled rovers, manipulator arms, and so on.

With increasing computational power made available by advances in microelectronic technology, computing infrastructures of robotic systems have recently evolved from single processor systems to networks of embedded systems [3], i.e. systems assuming that the computing resources are embedded into some other device including hardware and mechanical parts.

Smart sensors and actuators integrating sensing, actuating, processing, and networking elements into a single device simplify system architecture, by decreasing wiring requirements and improve modularity of the computing infrastructure by locally performing some signal processing tasks, but at the same time increase the complexity of software applications and make the development of reusable and portable components extremely challenging.

Several networking architectures can be used to interconnect such devices, including wired and wireless networks, or a combination of both. Supporting reliable Real-Time (RT) communication is one of the major requirements that are usually imposed to robot communication systems [4], where real-time control data must be periodically transferred between sensors, controllers, and actuators according to strict transfer deadlines.

Despite the semantic similarities between the operations supported by similar devices (e.g. all ranging provide distance measurements), the externally visible behavior of the software that abstracts and interfaces to each device greatly depends on device hardware architecture [5]. There are devices that have their analog and digital signals directly mapped to memory registers on the central processor. For these devices, all basic functionality (e.g. measurements filtering, image processing, PWM motor control, camera synchronization, etc.) are implemented in software, which thus requires dedicated computational, and synchronization resources. On the other hand, there are devices that implement much of their low level functionality in their firmware and thus reduce the load on the central processor. In both cases, the software that abstracts a physical devices should hide the details of the actual device but provide information about its usability (memory requirements, performance, realiability, etc.).

C. Application variability

Robots are situated agents and robot situatedness refers to existing in a complex, dynamic, and unstructured environment that strongly affects the robot behaviour. Situatedness implies that the robot is aware of its own internal state (e.g. resource availability) as well as of its temporal and spatial interactions with the environment. Sensing, actuating, and control of components may be subject to hardware failures or computational overload. The tremendous variety and open ended nature of human environments creates enormous challenges

to the system engineers ability to easily customize perception capabilities and sensory-motor skills (robustness by design) and the robot ability to exploit at best available resources (robustness by adaptation).

Over the past two decades, researchers have explored several control architectures for their robotic systems. They are typically classified as deliberative, reactive, behavior-based, and hybrid architectures [6]. Each architectural paradigm defines how the overall robot control system is partitioned into a set of sequential or concurrent control activities, from low-level control of motors and sensors to high-level capability such as planning and object recognition, and how these activities are amalgamated and coordinated.

In order to be broadly reusable, robotic software components have to support system integration according to different architectural paradigms. This can be achieved by designing individual components with the ability to interact not exclusively at a syntactical level, where components commonly agree on a set of data structure definitions and on the the operations to manipulate those structures, but rather at a semantic level, where components agree on the operational semantics of their interactions. This includes the components' roles in given contexts of use and the interaction protocols among components.

D. Separation of concerns in robotics component systems

In current practice, the design of a robot software architecture is mainly guided by the functional requirements of each specific robotic control application. While implementing robot functionality and control activities, the main concerns addressed by system enginneers are related to system performance, robustness, and dependability, which depend on the hardware setup, the robot's task and operational environment, and the robot-user interaction modes. This approach has lead so far to the development of highly successful robotic systems, but unfortunately has not enabled robot software reusability since most robotic applications are monolithic systems developed from scratch each time.

Many researchers are now facing the challenge of defining a new robot development process where complex systems are assembled from a set of reusable components developed for a family of similar applications. In the early stages of an application development, the designer has to take into consideration which components are available, which integration effort they require, and whether to reuse them as they are or build new components from scratch. According to Boehm [7], "in the old process, system requirements drove capabilities. In the new process, capabilities will drive system requirements...it is not a requirement if you can't afford it".

In order to maximize the reusability of common functionality implementation, software components should be designed having in mind the variability dimensions illustrated above and how these determine changes in the initial requirements of software components.

Supporting seamless evolution of a component-based robotic system with frequently changing requirements advocates for the separation of different design concerns, in such a way that component features affected by robot variability can be changed independently one from the others. These include the deployment of components on different and possibly networked computing platforms, the data exchange coordination and synchronization among components, the selection and composition of components providing specific functionality, and the assignment of computational resources to each component. The next four sections illustrate four software component design concerns as defined in [8], namely: Computation, Configuration, Communication, and Coordination.

III. Computation

Computation is concerned with the data processing algorithms required by an application [8].

Data processing algorithms are defined in terms of:

- data structure (what is manipulated by the algorithm)
- operations (how data are transformed by the algorithm)
- behavior (the order in which operations are performed on the data)

A software component is a computation unit that encapsulates data structures and operations to manipulate them. In order to enhance component reusability, it is convenient to separate the specification and implementation of data transformations and of control transformations. Data transformations are operations that elaborate data passed through input parameters, return values, and possibly update the component internal state. A control transformation is a specification for a finite-state machine (FSM), which activates or deactivates data transformations and is triggered by an execution thread [9].

This separation reflects the distinction between *data flow design*, where the main concerns are data availability, transformation, and streaming and the dominant question is how data moves through a collection of (atomic) computations from sources to destinations, and *control flow design*, where the main concerns are the order of computation of data transformations, serial or concurrent execution, failure or completion, synchronous or asynchronous processing, orchestration and the dominant question is how locus of controls moves through the program [10].

In this section, we focus on the computation model of software components by defining a set of architectural rules for encapsulating control transformations (at different levels of granularity) in order to carry out the set of transformation on the data.

In section VI, the rules of how control transformations are coordinated and a framework in which the interactions of individual components can be expressed will be presented.

A. Granularity of control transformations

In a complex system, several control transformations are executed concurrently and concurrency occurs at different levels of granularity. These are usually classified in fine, medium, and large grain as in [11]. In architectural perspective, by generalizing the Client/Server/Service desgn pattern [12],

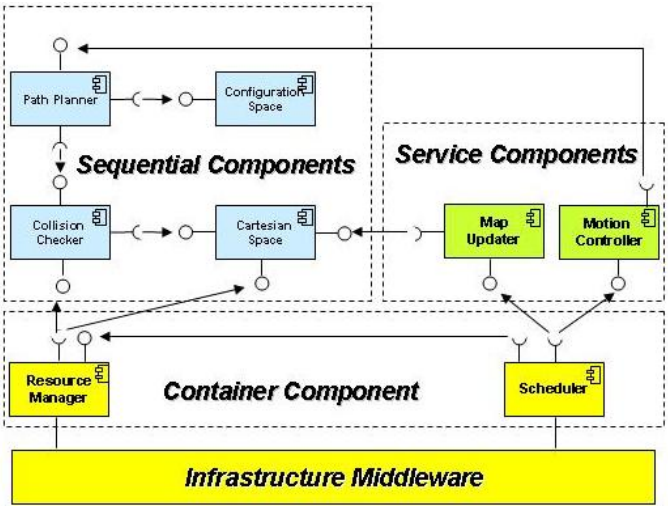

Fig. 1. The architecture of a Component Assembly

we map these levels of concurrency to three units of design respectively: the *Sequential Component*, the *Service Component*, and the *Container Component* (see Figure 1).

Sequential Components encapsulate data structures and operations that implement specific processing algorithms. Operations are short-lived actions without a predefined order (also called a weak cohesion), which depend on client threads of control for the execution of their functions and their behavioral semantics is guaranteed only in the presence of a single thread of control. As an example of this type of components, let's consider a path planner component, which implements a set of operations such as *planPath()*, *getNextConfiguration()*, and *changeViaPoint()*.

Service Components represent independent sequential execution threads of control. They offer the protected environment to fine grain concurrency, hide local data from other threads, maintain a symbolic state value, ERL:SOA2005and broadcast events with the incoming state name every time a state transition occurs. Typically, Service Components implement the logic and embed the dynamic specification of robot control activities, such as closing the loop between sensors and actuators for motion control or notifying the robot navigation subsystem when obstacles are detected.

Container Components provide the environment for the concurrent threads and encapsulate the shared resources. Container Components possess, create, and internally manage one or more independent threads of control that govern the execution of their services. They are structured as software frameworks composed of services that provide the runtime support to guarantee extra-functional properties by means of mechanisms for resource management, service scheduling, and Quality-of-Service (QoS) negotiation [13]. Figure 1 shows two services for (1) managing computational resources (*Resource Manager*) and (2) executing more than a single service concurrently (*Scheduler*). The Resource Manager implements mechanisms for admission and reservation of system resources, such as CPU time, memory, communication devices, and robotics devices (sensors, and actuators). The (optimal) allocation of resources to Service Components over time is a scheduling problem. The Scheduler has the capability of managing the temporal constraints regarding the real-time and non real-time execution of concurrent services.

Container Components support the so called "inversion of control": they act as clients of Sequential and Service Components and execute their operations when external events occur. For this purpose, Sequential and Service Components implement standard interfaces, i.e. for resource management and scheduling.

Container Components hide the interfaces of runtime services provided by the underlying infrastructure middleware from component developers, and instead automatically invoke them using developer-specified policies, thus eliminating coupling of Sequential and Service Components with the platform-specific environment and enhancing their reusability.

B. Components Assembly

Following the classification of software components based on the scope of their reuse illustrated in Part I of this tutorial, Sequential Components are typical examples of *vertical components*, i.e. they capture a research community know-how and thus contribute at most to software reuse. Container Components can be classified as *horizontal components*, i.e. they provide functionality to a variety of applications and can be customized by middleware experts. Finally, Service Components are mostly *application-specific components* as they represent the glue between the computational software infrastructure and the library of robot functionality and define the execution, interaction, and coordination of robot activities.

The architectural distinction in three different component types enforces the separation of variability concerns described in Section II and supports independent evolution of heterogeneous technologies. Container Components are to great extent middleware-specific (e.g. w.r.t. the support to threading, synchronization, and communication provided by the underlying runtime infrastructure) and functionality-independent. Sequential Components should conveniently be designed to be middleware- and application-independent. They may implement algorithms that, to some extent, are tightly coupled to specific robotic hardware, operational environment, and robotic task. In Section IV we will discuss how alternative implementations of the same Sequential Components can be selected even at run-time. Service Components are mostly functionality- and application-specific (i.e. they implement tasks that use specific robotic resources and must satisfy specific timing requirements).

A set of Sequential, Service, and Container Components form all together a Component Assembly. Sequential Components are typically deployed as binary files as they are intended to be reused, without changing the source code, within applications developed by third parties. Service Components implement the control logic of the application and are thus developed by adapting and extending a library of common robot control behaviors. Container Components are software frameworks that provide hot spots where Sequential and Service Components can be plugged-in.

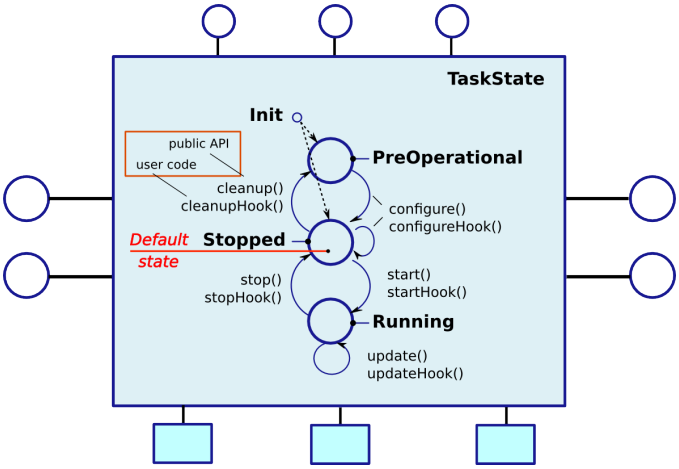

Fig. 2. OROCOS component state machine model.

Typically, Container Components are deployed as class libraries or binary files. In the former case (white-box reuse), the application developer is allowed to customize the Container's features when implementing Service Components, but requires the Container Component to be compiled with the custom code to create a Component Assembly. The latter case (black-box reuse) is advantageous when alternative implementation of Container Components are available from different providers, each one with specialized and optimized features (e.g. a sporadic server scheduling algorithm) and for specific infrastructure middlewares [14]. In this case, the Service Components are packaged in separate binary files and the application developer needs only to select the most appropriate Container Component for his application. Container Components need to be configured to automatically load, when instantiated, the selected Sequential and Service Components.

C. The OROCOS component model

The Open Robot Control Software (OROCOS) project [15] has developed a modular software framework for robot and machine control. The framework includes three main libraries: the Real-Time Toolkit (RTT) provides the infrastructure and the functionality to build component-based real-time applications; the Kinematics and Dynamics (KDL) library is an application independent framework for modelling and computation of kinematic chains; the Bayesian Filtering library(BFL) provides the software implementation of most used Bayesian methods.

OROCOS components encapsulate a single real-time thread that executes user-defined code. The OROCOS component model provides mechanisms for lock-free synchronous and asynchronous communication between real-time and non real-time threads and interfaces for distributed communication. Components communicate through interaction ports, which define the type of operations a component can perform. Depending on the type of active operation, a component can execute a particular action from its valid state according to the OROCOS finite state machine (Figure 2).

Giuliano Provvisionato

IV. Configuration

Configuration determines which system components should exist, and how they are inter-connected[8]

The software architecture of a complex system is described in terms of identified components, how they are connected, and the nature of the connectors (which specify interactions and communication patterns between the components).

Components are the principal units of computation. At architectural level, components are mainly described in terms of their provided and required interfaces.

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions [16]. In most component models, connectors do not correpsond to compilation units, but manifest themselves in different ways such as shared variables, initialization parameters, table entries, instructions to a linker, and so forth [17]. In other component models (e.g. [18], [19], [20]), connectors are first-class entities modeled and implemented as special components, which specifically deal with interactions and dependencies among components.

Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure [16]. The specification of a system configuration is conveniently kept separated from the specification of individual component behavior. This separation of concerns facilitates the description, comprehension, and manipulation, both by man and machine, of the system in terms of its structure [21]. In particular, explicit description of configurations enable assessment of extra-functional requirements (e.g reliability, performance, deadlock-free concurrence, etc.) both at design-time and run-time.

The goal of Component-Based Software Engineering is to enable useful integration of independently developed components into application-specific configurations.

Ideally, new applications are developed by reusing existing components only and interconnecting them by means of general purpose connectors. Practical experience in software engineering has demonstrated that effective reuse of software components can be achieved (1) if they adhere to the composition rules of a domain specific software architecture and (2) if their interconnections are flexible enough to accommodate the dynamic evolution of component-based systems due to changes in user requirements, upgrades of components, failure or substitution of devices, etc.

These two requirements have been addressed by the software engineering community in two research areas: Product Line Architectures and Configuration Programming. The former area studies how to design the family of possible configurations of a component-based system; the latter area studies how to design systems, whose configurations evolve dynamically.

A. Product Line Architectures

Component integration requires a common system architecture that defines not only how components interact by means of communication and synchronization mechanisms, but also the principles and policies that must be enforced by the set

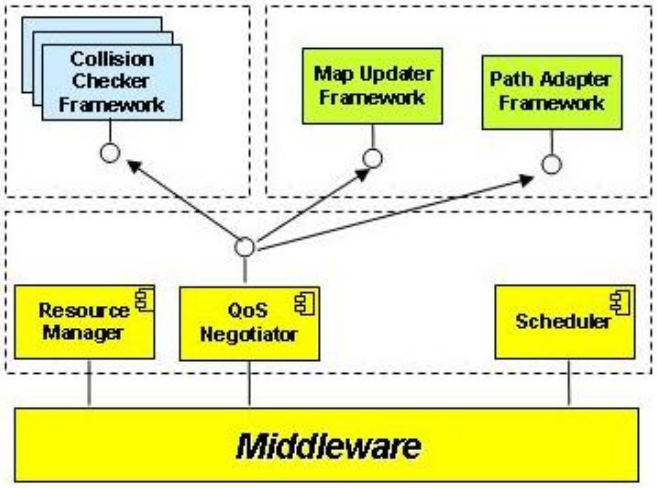

Fig. 3. Container services for dynamic component configuration

of interacting components. These include the role played by each component in the system, its provided and required interfaces, the interaction protocols, and the data structures used to represent, and exchange information (also called the information model).

A set of components, embedding functionality commonly found in a specific application domain and adhering to the principles of a common domain specific software architecture, form a so called Software Product Line [22].

The product line architecture specifies both commonalities and differences of a family of component-based systems. It defines the allowed variations that, when exercised, become individual products. For each variations, one or more component implementations are provided. Building software systems according to the product line approach is economic and efficient. Most work is about integration, customization, and configuration instead of creation.

A system configuration is thus an arrangement of components and associated options and settings that completely implements a software product [23]. Options may exclude each others (e.g. the selection of a component implementing an indoor navigation algorithm excludes the choice of components providing GPS-based localization services) or one option may make the integration of a second one a necessity (e.g. a component implementing a visual odometry algorithm depends on a component that supplies images of the surrounding environment). Hence only a subset of all combinations are admissible configurations. In [24] options are classified in five categories: *mandatory*, *optional*, *alternative*, *mutually inclusive*, and *mutually exclusive*.

Most variation points are bound at deployment time, but there may still be variation points that are bound only at runtime. This flexibility enable the dynamic reconfiguration of the component system according to the execution context.

B. Intra-Assembly configuration

The simplest example of component-based system is the Component Assembly. The configuration of Component Assemblies takes place at deployment-time, when binaries of Sequential and Service Components are loaded into the target Container Components.

The application developer builds Component Assemblies reusing off-the-shelf components (COTS) that meet at best functional and extra-functional requirements of the a robotic control system. In particular, he selects Container Components that are adequate to the underlying computational platform, the Sequential components that provide the required functionality (e.g. Motion Planning), and adapts tor develops Service Components that coordinate the execution of the component functionality. Several realizations of the each Sequential or Service Component specification can be deployed as part of the same Component Assembly. As an example, Figure 3 depicts the structure of the Motion Planner component assembly, which encapsulates a variety of Sequential Components implementing algorithms for collision checking with different extra-functional properties, such as efficiency, performance, completeness, and exception handling. These properties are also called software qualities. The quality of a component executing a service is called quality of service, or just QoS.

The quality of a component depends on both the specific implementation of the provided functionality and the availability of required resources. For example, the completeness of probabilistic motion planning algorithms increases with the processing time; similarly, the accuracy of collision detection algorithms increases with the amount of available memory; the timeliness of a decentralized control algorithm varies with the network bandwidth.

It is thus clear, that a component QoS depends on the execution environment, which is not known when the component is developed. At deployment time, the application developer can select components taking into account the worst case of resource usage, but the selection could be limited and the worst case assumption not applicable. A better approach differs the selection of the more adequate components at run time, when the actual execution context is known. The selection can then be based on the total amount of system resources, the set of available component implementations, and the most appropriate level of QoS for each component according to the application requirements.

A system is QoS-aware if it is able to trade quality in one dimension with another [25], e.g. completeness and efficiency. In architectural perspective, QoS-awareness is achieved by means of a QoS Negotiator Component which implements data structures and mechanisms for QoS management, and in particular:

- **QoS profile**. Every Sequential and Service Component implements the IQoS provided interface that allows the QoS Negotiatos to access the component's QoS profile. This profile consists of *offered* and *required quality statements* as well as the resource demand. In [13], quality statements are expressed in the CQML+ language as Boolean expression using current values of some system property (e.g. response time).
- **QoS negotiation**. The set of Component Assemblies in a system negotiate the quality level of the service provided by their constituent components, possibly involving the

cost for the service. The result of such a negotiation is a contract, which defines limits on the QoS characteristics that each service must operate within. In [13] the negotiation process is centralized and managed by a specialized component called the Contract Manager.

- **Service planning**. Each Component Assembly selects the best available implementation of Sequential and Service Components according to the QoS contract. Service planning [25] enables optimized sharing of resources between concurrent services.

The QoS negotiation and service planning processes are triggered by changes in the QoS profile of the components in the system. For example, motion planning in a static cluttered environment requires algorithms that guarantee completeness even if computational time demanding. In contrast, more efficient algorithms are needed for motion planning when the robot moves in dynamic environments. Consequently, the actual configuration of Component Assemblies change at run-time.

C. Inter-Assembly configuration

Complex systems are built by interconnecting different Component Assemblies at run-time, when their binaries are instantiated with configuration data that represent references to their communicating parties.

The interactions among Component Assemblies can change dynamically at run time according to the availability of their services, that can vary during their execution because of hardware failures (e.g. of a robotic device), limited network bandwidth, system overloading. In this context, configuration programming regards changing the state of a system as an activity performed at the level of interconnecting components rather than within the internal purely computational functionality of some component [26].

Several configuration and dynamic reconfiguration languages have been documented in the literature, such as Darwin/Regis [27], POLYLITH [28], and Rapide [29] to name a few.

In Wright [30] an architectural structure is represented as graph of components and connectors. Both components and connectors have interfaces, which represent points of interaction. Connectors identify the logical participants and encode the dynamism of the system configuration by explicitly identifying different patterns of interactions among components. For example, a connector may represent alternating configurations of a fault-tolerant Client/Server system, where multiple Server components offer redundant services (e.g. camera-based or sonar-based obstacle avoidance) and are dynamically switched in case of failures (e.g. the camera stops functioning). A special component, called *Configuror*, is responsible for achieving the changes to the architectural topology according to event generated by components notifying relevant state transitions, such as the activation or deactivation of component services.

More recently, Service Oriented Architectures (SOA) [31] have been proposed as an architectural paradigm, in which components are entities providing services described using a standard description language, the Web Services Description Language (WSDL). Applications are built dynamically by discovering services available on a computer network that provide required interfaces and assembling them on demand.

D. The ORCA component model

Orca is an open-source framework for developing component-based robotic systems [32]. It comes with a repository of stable and tested components for a variety of robot functionality and device drivers. In the context of ORCA, configuration is realized both on component and system levels. Components can be configured through a configuration file that specifies the appropriate algorithm implementation, the active communication ports and service interfaces. On system level configuration is dictated by inter-component connection topology, i.e. which components are communicating with each other. ORCA components are dynamically-loadable libraries that are deployed as services within the IceBox application server (http://www.zeroc.com/). A single conguration le species the set of components to instantiate and their individual conguration details. The IceGrid service enables start and stop of all components in the system.

V. Communication

Communication deals with the exchange of data [8].

Components of a robot control architecture have to communicate with each other in order to cooperate. Communication can be imperative, as in the case of a command issued by one component to another, or can be reactive, as in the case of event notification. Two communication mechanisms are deeply rooted in Object Oriented Programming, namely the Caller/Provider mechanism, which is involved when an object invokes another object method, and the Broadcaster/Listener mechanism, which gives objects the capability of broadcasting and listening to events.

One fundamental issue in communicating is that some sort of visibility [33] must exist among the parties. The communicating components, which are part of the same Component Assembly, reside in the same address space, thus the Caller holds a reference to the Provider (its memory address) to invoke its operations.

Visibility and flow of information can move in the same direction or in the opposite direction, with respect to the communicating components, according to the mechanism used to implement the communication. The Caller has visibility of the Provider and initiates the unidirectional (command) or bidirectional (query and response) exchange of information. The Listener has visibility of the Broadcaster, which notifies events to registered listeners.

Visibility implies dependency with respect to changes, and hence it has consequences on the reusability of components. In order to avoid the propagation of local changes to the entire architecture, visibility relationships have to be designed taking into account the possible evolution of the system and visibility loops should be avoided between components that evolve independently.

According to the component architecture described in the previous section, Sequential Components embed functionality

that are common to most robotic applications and implement stable and harmonized interfaces defined by robotics experts. In contrast, Service Components are more application-specific and are implemented by system integrators according to the functional requirements of each specific system. Thus, it is convenient to establish a communication pattern between a Service Component and a Sequential Component, where the former plays the role of Caller and/or Listener and the latter plays the role of Provider and/or Broadcaster.

A. Component decoupling

Service Components belonging to different Assembly Components are loosely coupled entities, which reside on rnetworked components, and whose relationships can change dynamically at runtime. The degree of decoupling between distributed Service Components can be analyzed along three dimensions that have been thoroughly described in [34], namely space decoupling, time decoupling and synchronization decoupling.

Space decoupling means that interacting components do not need to know each other, i.e. the Caller does not hold a reference to the Provider, and the Publisher does not know which subscribers are participating in the interaction. Space decoupling increases component reusability by removing explicit dependencies to specific components and system flexibility by supporting dynamic replacement and interconnection of individual components.

Time decoupling is enforced when the interacting components do not need to be actively participating in the interaction at the same time. Time decoupling enhances system reliability by making inter-component communication more robust to network failures.

Synchronization decoupling guarantees that callers, providers, publishers, and subscribers are not blocked while producing or consuming data and events. Asynchronous communication is a fundamental requirement of robotic control systems, where several activities process data (e.g. sensor measurements) concurrently and independently.

These kinds of decoupling are realized by means of services provided by Container Components that represent neutral mediators between interacting components. From an architectural perspective, such mediation services are usually represented by *Communication Ports* (see Figure 4).

Communication Ports rely on a middleware framework to exchange data and events through the network. Most common middlewares provide concrete implementations of various Communication Ports. Nevertheless, it is convenient to explicitly represent them at a more abstract architectural level because, in embedded systems, heavy-weight middlewares cannot be exploited due to systems limited resources and Communication Ports are more conveniently implemented using ad-hoc communication mechanisms, as described in [35]. In this case, the explicit representation of Communication Ports makes the properties of components' interaction more visible [36].

The following sections illustrate two well-known distributed communication paradigms that support, to different degrees, component decoupling along the three dimensions described above, namely *remote method invocation* and *publish/subscribe* [34].

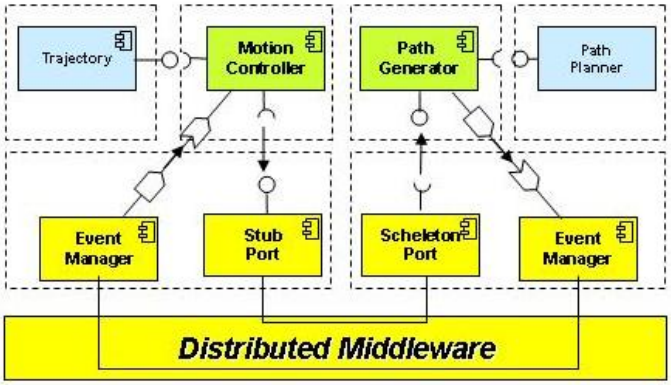

Fig. 4. Container services for distributed component interactions

B. Remote method invocation

The distribution paradigm adopted by some middleware frameworks like Object Management Group (OMG) common object request broker architecture (CORBA) [37] and Java remote method invocation (RMI) [38] consists in making remote invocations appear the same as local interactions. This is accomplished by means of two ancillary objects called the *stub* and the *skeleton*.

The stub is a surrogate (proxy) of the Provider and resides in the Caller's address space. It offers the same set of operations as the remote Provider. The Caller invokes the stub's operations as if it were the Provider itself. The stub is in charge of marshalling the Callers request and transmitting it through the network. The skeleton resides in the Providers address space, and is in charge of receiving and unmarshalling the Callers request and invoking the corresponding Providers operation. Similarly, the matched pair stub/skeleton is used to transmit the result of a Provider operation to the Caller through the network.

Figure 4 shows two Component Assemblies (the Caller on the left-hand side and the Provider at the right-hand side). The Motion Controller generates and controls the execution of robot trajectories. It delegates the definition of the corresponding motion paths to the Path Generator, which is hosted in a separate (possibly distributed) assembly. The Stub Port implements a *provided interface* that defines standard operations for common interaction patterns, such as *push(data d)*, *pull(data d)*, and *request(query q, response r)*. This operations are invoked by the Motion Controller to interact with the Path Generator. A corresponding set of operations are defined in the *required interface* of the scheleton port and are implemented by the Path Generator.

The remote method invocation paradigm imposes space coupling between the parties and is thus conveniently used for stable point-to-point interactions. This means that the Motion Controller interacts with a specific instance of Path Generator. It also introduces time coupling between the interacting parties

because both stub and scheleton services should be active at the same time. The synchronization decoupling depends on how the Provider's operations are implemented, namely as synchronous, asynchronous, or deferred-synchronous service requests. Synchronous requests block the Caller until the Provider has completed the execution of the requested operation and are typically used when a response from the Provider is required immediately or within some (application-specific) period of time, because a lack of response may prevent the Caller from continuing its execution. Asynchronous requests return immediately after the Provider has received the parameters of the invoked operation. Deferred-synchronous requests return a token that allows the Caller to retrieve the operation return value when needed.

C. Publish/Subscribe

The publish/subscribe interaction paradigm provides decoupling of distributed c omponents in all three time, space, and synchronization dimensions [34]. Messages, called events, are exchanged between publishers and subscribers in a asynchronous manner without the need for the interacting parties to know each others and to participate to the interaction at the same time.

An event is typically characterized by three information:

- The *type* characterizes events in such a way that subscribers can recognize the events they are interested in. Typically, it corresponds to a information subject that is identified by one or more keywords. For examples, the Navigator component is interested in *alarm* events indicating that an unexpected obstacle has been detected along the robot path. Every device component processing sensory data has to be programmed to notify this type of event.
- The *payload data* contains application-specific information. For example, a publisher component (e.g. the Motion Controller) may issue an event that indicates a change in its internal state (e.g. *motion done*).
- The *timestamp* indicates the instant, when the event has been generated.

Optional information may indicate the *priority* of the event, the *issuer* identifier, and the *expiration time*.

Full decoupling is achieved by means of Communication Ports that implement store and forward mechanisms both in the publisher's and subscriber's containers, so that communication appears asynchronous and anonymous to interacting Service Components.

The publisher posts events in the local container's Communication Port, which asynchronously (i.e. by means of a dedicated thread) forwards them to the Communication Port of registered subscribers. When an event is received, an event handler notifies the subscribers asynchronously. In case of network failures or unavailability of a subscriber, the publisher's Communication Port stores the event until it can be notified correctly.

For example, in Figure 4, the Path Generator plays the role of Publisher and notifies the Motion Controller (the subscriber) that the generation of a new path has been completed.

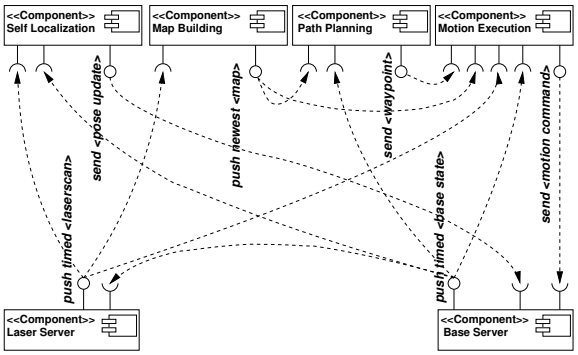

Fig. 5. The SMARTSOFT communication patterns.

Subscriptions can be handled in two ways. If the publisher's Communication Port maintains the list of subscribers to its events, it can establish a point-to-point connection with each subscriber when an event needs to be notified. In this case, publishers advertise the type of event they intend to notify in order to inform interested subscribers, which register themselves with the publisher. Alternatively, the publisher's Communication Port send events through the network using hardware multicast facilities like IP multicast. Every subscriber in the subnetwork receives the event, which is filtered by the local Communication Port according to the subscriber's criteria.

D. The SMARTSOFT component model

SMARTSOFT [39] is a project that specifically addresses issues related to communication among software components of a robotic control system. It aims at simplifying the integration of complex systems by defining a limited set of communication patterns typically found in robotic control applications. A communication pattern is an interface of a component with strictly defined interaction semantics.

SMARTSOFT defines the following set of generic patterns:

- *send* - defines one-way communication with client/server relationship
- *query* - two-way request communication with client/server relationship
- *push newest* - 1-to-n distribution (broadcast) with publisher/subscriber relationship
- *push timed* - 1-to-n distribution (broadcast) with publsher/subscriber relationship
- *event* - asynchronous conditioned notification with client/server relationship
- *dynamic wiring* - dynamic component wiring with master/slave relationship

These are sufficient since they cover request/response interaction as well as asynchronous notifications and push services. Communication patterns enforce standardized service contracts between loosely coupled components (Figure 5). The interaction patterns can be enriched with resource information and timing constraints such that the deployment of components can be cross-checked against capabilities of the target platform.

VI. COORDINATION

Coordination is concerned with the interaction of the various system components.[8]

Service Components embed the control logic of a robot control application. They cooperate with each other locally or through a communication network to achieve a common goal according to the robot control architecture and compete for the use of shared resources, such as the robot sensors and actuators, the robot functionality offered by Sequential Components, and the computer processing and communication resources.

Cooperation and competition are forms of interactions among concurrent activities, implemented as sequential threads of control encapsulated inside Service Components. Concurrency means that computations in the system overlap in time [40] and are interleaved with one another on a single processor. Correct interleaving of concurrent activities is governed by algorithms called interaction protocols [41], such as mutual exclusion protocols, which ensure that a non-sharable resource is only used by one activity at a time and that an activity which wants to access the non-sharable resource will get access to it eventually.

Interaction protocols are implemented using synchronization mechanisms such as locks, semaphores, and event handlers. An event handler, in an event-driven control environment, is the most commonly used synchronization mechanism. It blocks the current thread until an event is received from one of the objects it monitors. Every time a thread calls the waitEvent(Symbol filter) method of the event handler, it is suspended until the event named filter is raised. When one of the events to which the event handler is attached is raised, the threads waiting for this event are awakened in turn and continue their execution.

Modeling and designing complex concurrent systems is a difficult task, which is even more exacerbated in component-based system design, where abstraction, encapsulation, and modularity are main concerns. Components encapsulate, and hide to the rest of the system, how computations are ordered in sequential threads and how and when computations alter the system state. The consequence of improper management of the order and containment relationships in a complex, concurrent system is deadlock [42].

A. Coordination models and languages

In order to alleviate the difficulties of designing complex concurrent systems, coordination languages and models have been proposed as valid tools to managing the interaction among concurrent components [26].

Coordination languages and models fall into one of two major categories of coordination programming, namely either data-driven or control-driven [26]. Data-driven coordination models define the state of the computation of the entire system at any moment in time in terms of the actual system configuration and the data exchanged by system components through a shared tuple space [43]. A tuple is a an ordered list of structured data, has an existence which is independent of its producer, and it is equally accessible to all components until it is explicitly withdrawn by some component. Typical examples of data-oriented applications are transactional information systems. In contrast, control-driven coordination models define the state of the computation of the entire system at any moment in time in terms of only the current internal state of each component. Components observe state transitions in the systems by listening to events notified by other components.

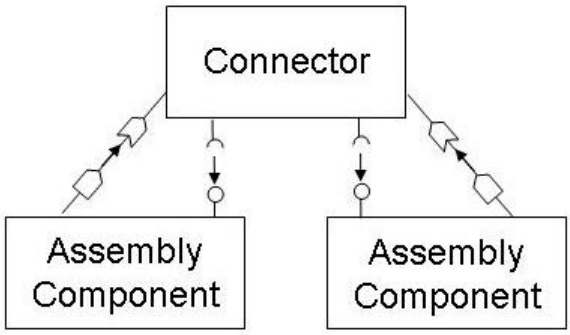

Fig. 6. Architectural relathionships between Components and Connectors

The two categories of coordination languages also differ in terms of programming style. Data-driven languages like LINDA [44] generally lead to intermixing of coordination primitives with computation code, making the coordination model and the coordination protocol implicit. In contrast, control-driven models enable a clear and complete separation of coordination and computation concerns. This is usually achieved by using different languages for expressing computation and coordination and by developing coordination components separately and independently of the computation components they coordinate. An example of coordination language is Manifold [26], a strongly-typed, block-structured, event-driven language, meant for writing coordinator program modules.

B. Connectors as coordination components

Recently, the relations between software architectures and coordination models have been explicitly studied [45] with the goal of profitably and coherently exploit both approaches to component-based system design.

Connectors represent first class architectural entities embodying component interaction. For example, the UniCon language [46] define a set of predefined connector types that correspond to common communication primitives. In the Wright language [47], the interactions among components are fully specified by the user and complex interactions can be expressed by nested connector types.

The taxonomy of software connectors presented in [48] identifies and analyzes the basic tasks a connector should perform, such as:

- Interface adaptation and data conversion for components that have not been originally designed to interoperate.
- Control and data transfer between local and remote components.
- Communication intercepting for implementing various

filters (e.g. cryptography, data compression, load monitoring, etc.).
- Access coordination and ordering of method calls on a component interface.

The connector task indicated in the last item is particularly relevant to the discussion on component coordination. The permitted orderings of method calls on a components interface are usually determined by a behavioral specification of the component, which represents the contract between a component and its clients and assumes the form of a coordination protocol. For example, a coordination protocol may specify that two operations of a component interface must be invoked immediately one after the completion of the other.

Several formalisms have been proposed to specify interaction protocols, such as abstract states and state transitions [49], process algebra-based specification [47], pi-calculus [50], and algebraic constraints on the relevant interface elements [29].

Coordination connectors are in charge of enforcing complience with the protocol of a set of interfaces, mediating clients access to the component. In [20] connectors completly encapsulate control and data flow between connected components. This menas that components do not request services in other components, but perform their provided services only when invoked externally by connectors. In architectural perspective, the interconnection of components and connectors can be described as in Figure 6: a connector plays the role of *subscriber* to the events published by two or more connected components and of *caller* of the operations defined in their provided interface. Connectors represent the mean to achieve compositionality of individual software components into complex component systems. In [20] connectors are organized into hierarchies of composition, where lower level connectors behave as computational components and are interconnected by higher level connectors. In [51] a completely decentralized approach to composition is presented.

C. The GenoM/BIP component model

GenoM [52] is a component-oriented software package developed by LAAS CNRS robotics group. Components interface hardware devices or encapsulate common robotics algorithms. GenoM components (see Figure 7) are collections of control services, which manage incoming requests, and execution services, which implement specific algorithms. Every incoming request is represented with a finite state automaton and is implemented by a set of C functions called *codels*, which get appropriatey called during specific state transitions (i.e. start, exec, error, etc.). One of the most important property of codels is that they are non-preemtive as soon as they are active. When a codel is executed in a thread all the resources that it uses are released after specific period of time (the length of this period is defined in module specification file) regardless of results of the execution. GenoM components exchange data and events through *'posters'* which are regions of shared memory. System level coordination is achieved by means of the Behaviors Interactions Priorities (BIPs) framework. BIP is used to produce a formal interaction model, which can be used to run (using the BIP engine) the functional layer composed

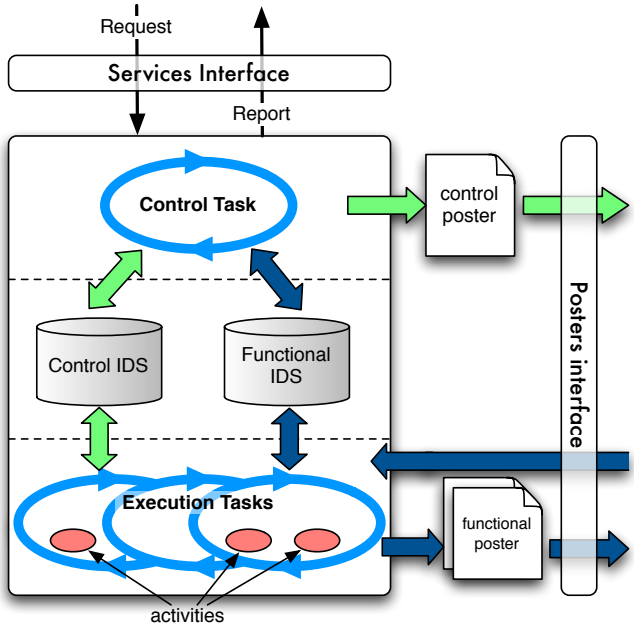

Fig. 7. The internal structure of a GenoM component

of all the genom modules. BIP allows hierarchical construction of compound components from GenoM components by using connectors and priorities. A connector interconnects the ports of GenoM components and models two basic modes of communication, namely synchronous and broadcast. Priorities reduce nondeterminism in componnet interactions.

VII. Conclusions

In this second of a two-parts tutorial on Component-Based Robotic Engineering we have illustrated design guidelines to decompose the functionality of complex robotic control systems into different types of components and techniques to assemble them into large and distributed component systems. The focus was on the architectural models that enable the independent evolution of different variability concerns of a component-based robotic system.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics).

The authors would like to thank Felix Ingrand, Patrizia Scandurra, Christian Schlegel, and all the partners of the BRICS project for their valuable comments.

References

[1] D. Brugali and E. Prassler, "Software engineering for robotics," Special Issue of the IEEE Robotics and Automation Magazine, March 2009.
[2] A. Diaz-Calderon, I. A. D. Nesnas, H. D. Nayar, and W. S. Kim, "Towards a unified representation of mechanisms for robotic control software," *International Journal of Advanced Robotic Systems*, vol. 3, pp. 61–66, 2006.

[3] B. Ge, G.Yasuda, and H. Takai, "A microcontroller-based architecture for locally intelligent robot agents," in *Fifth World Congress on Intelligent Control and Automation*, vol. 6, June 2004, pp. 4826 – 4828.

[4] B. Baeuml and G. Hirzinger, "When hard realtime matters: Software for complex mechatronic systems," *Robotics and Autonomous Systems*, vol. 56, pp. 5–13, 2008.

[5] I. A. Nesnas, "The claraty project: Coping with hardware and software heterogeneity," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Berlin / Heidelberg: Springer - Verlag, April 2007, vol. 30.

[6] M. J. Mataric, "Situated robotics," *Encyclopedia of Cognitive Science, Nature Publishing Group, Macmillan Reference Limited*, 2002.

[7] B. W. Boehm, *Software Engineering Economics*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[8] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *Proceedings of the Int, Workshop on Trends in Distributed Systems CORBA and Beyond*. Springer LNCS, 1996, vol. 1161, pp. 162–176.

[9] K.-K. Lau and F. M. Taweel, "Domain-specific software component models," in *CBSE '09: Proceedings of the 12th International Symposium on Component-Based Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 19–35.

[10] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[11] R. S. Chin and S. T. Chanson, "Distributed, object-based programming systems," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 91–124, 1991.

[12] A. Aarsten, D. Brugali, and G. Menga, "Designing concurrent and distributed control systems," *Commun. ACM*, vol. 39, no. 10, pp. 50–58, 1996.

[13] S. Göbel, C. Pohl, R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler, "The COMQUAD component container architecture," in *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, J. Magee, C. Szyperski, and J. Bosch, Eds., Jun. 2004, pp. 315–318.

[14] G. A. Moreno, "Creating custom containers with generative techniques," in *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*. New York, NY, USA: ACM, 2006, pp. 29–38.

[15] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the orocos project," in *Proceedings of the 2003 IEEE International Conference on Robotics and Automation, ICRA 2003, September 14-19, 2003, Taipei, Taiwan*. IEEE, 2003, pp. 2766–2771.

[16] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[17] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314–335, 1995.

[18] A. Amirat and M. Oussalah, "Connector based metamodel for architecture description language." *INFOCOMP Journal of Computer Science*, vol. 8, no. 1, pp. 55–64, 2009.

[19] D. Bálek and F. Plasil, "Software connectors and their role in component deployment," in *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 2001, pp. 69–84.

[20] K.-K. Lau, L. Ling, P. Velasco Elizondo, and V. Ukis, "Composite connectors for composing software components," in *Proc. 6th Int. Symp. on Software Composition, LNCS 4829*, M. Lumpe and W. Vanderperren, Eds. Springer-Verlag, 2007, pp. 266–280.

[21] J. Kramer, "Configuration programming–a framework for the development of distributed systems," in *IEEE Internat. Conf. on Computer Systems and Software Engineering*, Israel, May 1990, pp. 374–384.

[22] C. T. R. Lai and D. M. Weiss, *Software Product-Line Engineering: A FamilyBased Software Development Process*. Addison-Wesleyy, 1999.

[23] M. S. Sijbren Keimpe Deelstra, "Managing the complexity of variability in software product families," Ph.D. dissertation, RIJKSUNIVERSITEIT GRONINGEN, 2008.

[24] M. Anastasopoulos and C. Gacek, "Implementing product line variabilities," *Technical report IESE report N089.00/E, Franhofer IESE publication, 2000*, May 2001.

[25] O. M. Wergeland, "Service planning in a qos-aware component architecture," Ph.D. dissertation, University of Oslo, 2007.

[26] G. A. Papadopoulos and F. Arbab, "Coordination models and languages," *Advances in Computers*, vol. 46, pp. 330–401, 1998.

[27] J. Kramer, J. Magee, and A. Finkelstein, "A constructive approach to the design of distributed systems," in *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*. IEEE Computer Society, 1990, pp. 580–587.

[28] J. M. Purtilo, "The polylith software bus," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 1, pp. 151–174, 1994.

[29] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using rapide," *IEEE Trans. Software Eng.*, vol. 21, no. 4, pp. 336–355, 1995.

[30] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[31] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[32] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Orca: a component model and repository," in *Software Engineering for Experimental Robotics*, ser. STAR, D. Brugali, Ed. Springer Berlin / Heidelberg, 2007, vol. 30/2007, pp. 231–252.

[33] G. Booch, *Object-Oriented Design with Applications*. Benjamin/Cummings, 1990.

[34] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[35] P. Corke, P. Sikka, J. Roberts, and E. Duff, "Ddx: A distributed software architecture for robotic systems," in *Proc. Australian Conf. Robotics and Automation*, Canberra, December 2004.

[36] D. Schreiner and K. M. Göschka, "Explicit connectors in component based software engineering for distributed embedded systems," in *SOFSEM '07: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 923–934.

[37] "Omg corba," http://www.corba.org/ (last access: 3rd December 2009).

[38] "Java remote method invocation," http://java.sun.com/javase/technologies /core/basic/rmi/index.jsp (last access: 3rd December 2009).

[39] C. Schlegel, "Communication patterns as key towards component interoperability," in *Software Engineering for Experimental Robotics*, ser. STAR, D. Brugali, Ed. Springer, 2007, vol. 30/2007, pp. 183–210.

[40] F. Arbab, "What do you mean, coordination?" Bulletin of the Dutch Association for Theoretical Computer Science, Tech. Rep., March 1998.

[41] H. Rust, "Concurrency and reactivity: Interleaving," in *Operational Semantics for Timed Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, ch. 8, pp. 75–82.

[42] J. S. Davis, II, "Order and containment in concurrent system design," Ph.D. dissertation, 2000, chair-Lee, Edward A.

[43] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[44] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[45] P. Inverardi and H. Muccini, "Software architectures and coordination models," *J. Supercomput.*, vol. 24, no. 2, pp. 141–149, 2003.

[46] M. Shaw, R. DeLine, and G. Zelesnik, "Abstractions and implementations forarchitectural connections," in *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1996, p. 2.

[47] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Trans. on Software Engineering and Methodology*, July 1997.

[48] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA: ACM, 2000, pp. 178–187.

[49] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 2, pp. 292–333, 1997.

[50] C. Canal, L. Fuentes, J. Troya, and A. Vallecillo, "Extending corba interfaces with p-calculus for protocol compatibility," *Technology of Object-Oriented Languages, International Conference on*, p. 208, 2000.

[51] F. Balduzzi and D. Brugali, "A hybrid software agent model for decentralized control," in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation, ICRA 2001, May 21-26, 2001, Seoul, Korea*. IEEE, 2001, pp. 836–841.

[52] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and N. Thanh-Hung, "Designing autonomous robots," *IEEE Robotics and Automation Magazine*, vol. 16, pp. 67–77, March 2009.