Project IST-2001-34820

ARTIST

Advanced Real-Time Systems

ROADMAP

Component-based Design and Integration Platforms

W1.A2.N1.Y1

Year 1 Reference Period: April 1st 2002 – March 31st 2003

Review: May 6th, 2003

In this roadmap, we survey techniques for handling different functional and extra-functional properties of component and system behavior. A conclusion is that techniques exist for handling such properties, but that further work is needed to improve the theory of specifying and composing components, and to develop tool support.

Contributors

Ed Brinksma

Geoff Coulson

Ivica Crnkovic

Andy Evans

Sébastien Gérard

Susanne Graf

Holger Hermanns

Bengt Jonsson

Anders Ravn

Philippe Schnoebelen

Francois Terrier

Angelika Votintseva

Jean-Marc Jézéquel

Contents

0 Executive Overview **2**

1 Introduction **4**

1.1 Basic Motivation . . . 4

1.2 Basic Concepts . . . 5

1.3 Concepts Related to Component Technology . . . 6

1.4 Structure of This Document . . . 7

2 Component-Based System Development **7**

2.1 Life cycle of Component-based Systems . . . 8

2.2 Life Cycle of Components . . . 10

2.3 Issues Specific for Embedded Systems . . . 11

2.4 Summary and Conclusions . . . 12

3 Situation and Needs in Different Industry Sectors **13**

3.1 Automotive . . . 13

3.2 Industrial Automation . . . 14

3.3 Consumer Electronics . . . 16

3.4 Telecommunication Software Infrastructure . . . 17

3.5 Avionics and Aerospace . . . 18

3.6 Summary and Conclusion . . . 19

4 Specifying and Reasoning about Contracts **21**

4.1 Introduction . . . 21

4.2 Level 1 - Syntactic Interfaces . . . 22

4.3 Functional Properties (Level 2) . . . 24

4.4 Functional Properties (Level 3) . . . 26

4.5 Timing Properties (Level 4) . . . 28

4.6 Level 4 - Quality of Service . . . 32

4.7 Reliability . . . 36

4.8 Specifying and Reasoning about Contracts: Summary and Analysis . . . 36

5 Component Models and Integration Platforms: Landscape **37**

5.1 Component Models for Embedded System Design. . . . 38

5.1.1 Programmable Logic Controllers: the IEC 61131-3 standard . . . 38

5.1.2 Koala . . . 40

5.1.3 Rubus Component Model . . . 43

5.1.4 PECOS . . . 44

5.2 Component Models for Run-Time Composition 45
5.2.1 Java Beans 45
5.2.2 EJB - Enterprise Java Beans 46
5.2.3 COM, DCOM, COM+ 48
5.2.4 .NET 50
5.2.5 CORBA and CCM 51
5.2.6 Real-time CORBA 54
5.2.7 Analysis 54
5.3 Integration Platforms for Heterogeneous System Design 55
5.3.1 MetaH 56
5.3.2 Ptolemy II 58
5.3.3 Metropolis 59
5.4 Hardware/Software Modeling Languages 60
5.4.1 System C 60
5.4.2 VHDL 60
5.5 Component Models and Integration Platforms: Summary and Conclusions 60

6 Findings, Synthesis, Needs **61**

7 Missing pieces **62**

8 Clearly-Identified Priorities **62**

9 Standardization Efforts **63**
9.1 Specification Standards 63
9.1.1 UML 2.0 63
9.1.2 Components at the Model Level 64
9.1.3 Real-time UML Profile 64
9.1.4 EDOC 65
9.1.5 MDA 66
9.1.6 MDA & current industrial Real-time-UML tools 66
9.1.7 Towards MDA components? 67
9.2 Implementation Technology Standards 68
9.2.1 SOAP 68
9.3 Conclusions and Challenges 68

0 Executive Overview

Component based design is seen to increase software productivity, by reducing the amount of effort needed to develop, update, and maintain systems. There are two main benefits specific to component technology. First, it gives structure to system design and system development, thus making system verification and maintenance more tractable. Second, it allows reuse of development effort by allowing components to be re-used across products and in the longer term by paving the way for a market for software components.

Roughly speaking, there are three lines of widely adopted component technologies: JavaBeans/EJB from Sun. COM/DCOM/COM+/.NET from Microsoft, and and CORBA/CCM from OMG. Early component technologies such as JavaBeans, COM, and CORBA offer protocols whereby independently developed components can be deployed and collaborate. The technologies have subsequently developed into sophisticated platforms (e.g., EJB, .Net, CCM) which offer a variety of runtime services for managing component activation, concurrency, security, persistency, distribution, and transactions.

Adoption for the development of real-time systems is significantly slower. Major reasons are that real-time systems must satisfy requirements of timeliness, quality-of-service, predictability, that they are often safety-critical, and can use severely constrained resources (memory, processing power, communication). The widely adopted component technologies described in the preceding paragraph are inherently heavyweight and complex, incurring large overheads on the run-time platform; they do not in general address timeliness, quality-of-service or similar extra-functional properties that are important for real-time systems. In their present form they start to be deployed in large, distributed, and not safety critical systems, e.g., in industrial automation, and are not suitable for deployment in most embedded real-time environments.

For small real-time systems, component technologies have been developed for particular classes of systems. Often, these have been done within development organizations, and their adoption outside these organizations is limited. To avoid heavy-weight run-time platforms, they mostly do not support run-time deployment of components and lack many services. Composition of components into a (sub)system is rather performed in the design environment, prior to compilation, thus enabling static prediction of system properties and global optimizations.

In this document, we survey the use of component technologies in different industrial sectors. Two important obstacles to wider adoption of component technology for real-time systems are the following.

- There is a lack of widely adopted standards for component technology. A complicating factor is that different sectors have different priorities concerning the main characteristics offered by such a standard.
- A component technology for real-time systems should support specification and prediction of timing and QoS-properties. Solutions to these problems are not well enough developed and not well enough integrated into development tools.

In the document, we survey techniques for handling different functional and extra-functional properties of component and system behavior. A conclusion is that techniques exist for handling such properties, but that further work is needed to improve the theory of specifying and composing components, and to develop tool support.

In a subsequent section, we survey technical characteristics of main component technologies. Two trends can be discerned: one is to define component technologies for specialized real-time applications, maybe building on existing RTOSs or simply deployed on target hardware. Another is to adapt widely adopted component technologies to embedded systems, by omitting functionality, and perhaps building efficient run-time platforms.

A conclusion is that in the future, we need widely adopted component technologies with common interfaces, which can be tuned and configured to support needs of particular application domains.

1 Introduction

1.1 Basic Motivation

Component-Based Development is perceived as key for developing advanced real-time systems in a both cost- and time effective manner. It can be seen a qualitative jump in software development methodology, comparable to the transition from assembly language programming to high level problem oriented languages around 1970, or the transition from procedural programming to object oriented programming around 1990.

Component based design is seen to increase software productivity, by reducing the amount of effort needed to develop, update, and maintain systems. Benefits include the following.

- **Giving Structure to System Development.** Component technology supports the structuring of complex systems early in the development process. In this way, many integration and maintenance problems can be addressed early, at lower cost.
- **Reuse of Development Effort.** Components can be re-used across several products or even product families. Re-use is made easer by defining *product line architectures*, in which components have given roles. New products can then re-use components of previous products by slight modification or parameterization.
- **Supporting System Maintenance and Evolution.** Systems are easier to maintain if they have a clear structure, e.g., as a system composed of components. For legacy systems, it sometimes pays off to decompose them into components in order to make future upgrades and maintenance easier.
- **Enabling a Market for Software Parts.** Standardized component specifications and technologies allow to integrate components produced by different suppliers. Currently, for embedded systems, only large components are transferred between different organizations: RTOS, databases, process control components, etc. If a wider class of components would be re-usable across a wider class of systems, this would give higher returns on development investment. One vision for the future is that application development could follow a "drop & glue" approach, picking components from a library incorporating the intellectual property of the system house, as well as standardized components, giving to the system developer a range of re-usable components supporting all layers of a system architecture. This vision includes an open market of components, which are interoperable, and where integration problems are solved *a priori*.

Component technology has gained wide adoption in the area of business data processing, and is under continuous development. There are also signs of adoption for the development of embedded and real-time systems. However, the pace is significantly slower, major reasons being that other concerns are of great importance for the development of such systems. Real-time systems must satisfy constraints on extra-functional properties such as timing (e.g., meeting deadlines), quality of service (e.g., throughput), and dependability (including reliability, safety, and security). It is important that functional and extra-functional properties be predictable, in particular if the system is safety-critical. Embedded systems must often operate with scarce resources (including processing power, memory, communication bandwidth). These concerns are not addressed by the most widely used component technologies.

Several component technologies have been developed that address the specific needs of embedded systems. There are many challenges to overcome in order to develop component technology that is suitable for the many particularities of embedded systems.

1.2 Basic Concepts

There is some disagreement about the precise definition of basic terms in component based software development. We therefore give a short treatment of basic ideas, and define how basic terms will be understood in this document.

A basic idea in component based software development is to structure a system into *components*. In classic engineering disciplines, a component is a self-contained part or subsystem that can be used as a building block in the design of a larger system. It provides specific services to its environment across well-specified interfaces. Examples are an engine in an automobile, or the heating furnace in a home. Ideally, the development a component should be decoupled from the development of the systems that contain is. Components should be reusable in different contexts.

In software engineering, there are many different suggestions for precise definitions of components in component based software development. According to [BBB+00], advocates of software reuse equate components to anything that can be reused; practitioners using commercial off-the-shelf (COTS) software equate components to COTS products; software methodologists equate components with units of project and configuration management; and software architects equate components with design abstractions.

The best accepted definition in the software industry world is based on Szyperski's work [Szy98]:

> a component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition.

In this report, we largely follow the definition by Szyperski, and in particular stress the separation between component implementation and component interface. Ideally, there should be no context dependencies that are not captured by the component interface. This last sentence must be applied with some care, since in practice typical interfaces capture only certain aspects of a component's behavior.

Szyperski [Szy98] tends to insist that components should be delivered in binary from, and that deployment and composition should be performed at run-time. In this report, we take a more liberal view, and consider a component as a software implementation that can be executed on a physical or logical device. This includes components delivered in high-level languages, and allows build-time (or design-time) composition. This more liberal view is partly motivated by the embedded systems context, as will be discussed in Section 2.3.

There are two basic prerequisites that enable components to be integrated and work together.

- A *component model* specifies the standards and conventions that components must follow to enable proper interaction.
- A *component framework* is the design-time and run-time infrastructure that manages resources for components and supports component interactions.

There is an obvious correspondence between the conventions of a component model and the supporting mechanisms and services of a component framework.

Component models and frameworks can be specified at different levels of abstraction.

- Some component models (e.g., COM) are specified on the level of the binary executable, and the framework consists of supporting OS services.
- Some component models (e.g., JavaBeans, CCM, or .Net) are specified on the level of byte-code.
- Some component models (e.g., Koala) are specified on the level of a programming language (such as C). The framework can contain "glue code" and possibly a runtime executive, which are bundled with the components before compilation.

In component based system development, there is a clear distinction between two perspectives of a component.

- The component *implementation* is the executable realization of a component, obeying the rules of the component model. Depending on the component model at hand, component implementations are provided in binary form, byte code, compilable C code, etc.
- The component *interface* summarizes the properties of the component that are externally visible to the other parts of the system, and which can be used when designing the system. An interface may list the signatures of operations, in which case it can be used to check that components interact without causing type mismatches. An interface may contain additional information about the component's patterns of interaction with its environment or about extra-functional properties such as execution time; this allows more system properties to be determined when the system is first designed. An interface that, in addition to information about operation signatures, also specifies functional or extra-functional properties is called a *rich interface.*

The component implementations must of course conform to the properties stated in their interfaces. In principle this presupposes that the semantics of what it means for a component implementation to conform to information in its interface is well understood, and that there are mechanisms for checking or enforcing conformance, such as verification (simulation, testing, run-time monitoring, formal verification, etc.) and code generation.

The information in component interfaces facilitates the check for interoperability between components. Rich interfaces enable *verification* of system requirements and *prediction* of system properties from properties of components. This allows several system properties to be verified and predicted early in the development life cycle, enables early design space exploration, and saves significant effort in the later system integration phase. A research challenge today is to develop methods for predicting system properties from component properties.

A *contract* is a specification of functional or extra-functional properties of a component, which are observable in its interface. A contract can be seen as specifying constraints on the interface of a component.

1.3 Concepts Related to Component Technology

Architecture Description Languages. The *software architecture* of a program or computing system is generally taken to denote

> "the structure or structures of the system, which comprise software components [and connectors], the externally visible properties of those components [and connectors] and the relationships among them." [BCK98]

The architecture of a system is an early design decision, which to a large extent determines global system parameters such as functionality, performance, resource consumption, maintainability, etc. Descriptions of system architectures include descriptions of component properties, visible through their interfaces, and enable informed evaluations of different system architectures when selecting between them. *Architecture Definition Languages* (ADLs) have been developed as languages for expressing system architectures as compositions of software modules and/or hardware objects. Typical concepts of ADLs are *components*, *ports*, *connectors*, etc. They can also describe various classes of component properties. When used in component-based development, component properties expressed in a system description using an ADL should in principle also be expressible in component interfaces. For example, MetaH may decorate components with properties such as execution time and failure modes. Component interfaces must then be rich enough to allow description of such properties.

ADLs concentrate on the description of system, whose properties are the composition of properties visible in component interfaces. In contrast, a component technology must also specify how such interfaces are implemented (possibly from independently developed components), so that the resulting system implementation has the properties described in its architecture. Since the purpose of this document is to concentrate on components themselves, we refrain from giving an extensive overview of ADLs. A few ADLs that are perceived as influencing the development of component technology, are described in Section 5.

Software vs. System Components. Many important properties of components in embedded systems, such as timing and performance, depend on characteristics of the underlying hardware platform. Kopetz and Suri [KS03] propose to distinguish between software components and system components. Extra-functional properties, such as performance, cannot be specified for a *software component* in isolation. Such properties must either be specified with respect to a given hardware platform, or be parameterized on (characteristics of) the underlying platform. A *system component*, on the other hand, is defined as a self-contained hardware and software subsystem, and can satisfy both functional and extra-functional properties.

1.4 Structure of This Document

This document is structured as follows. In Section 2, we present a view on the development of component-based systems, as a basis for identifying key concerns for component based development, in particular for embedded systems. Section 3 presents condensed reports on the state of the art, trends, and needs for component based development in different industrial application sectors. In Section 4, we concentrate on presenting techniques used for specifying and analyzing important functional and extra-functional propeties of systems using information about component interfaces. Section 5 presents major component models, and assesses some of their strengths and limitations, in particular with respect to the aspects discussed in Section 4. Finally, in Section 6, we survey the situation with respect to standardization efforts, in particular related to OMG, that are central to component technologies for real-time systems.

2 Component-Based System Development

Component-based software engineering (CBSE) uses methods, tools and principles of general software engineering. However there is one distinction: CBSE distinguishes **component development** and **system development with components**. There is a slight difference in the requirements and business goals in the two cases and there exist different approaches.

- In **component development**, the main emphasis is on *reusability:* components are built to be used and reused in many applications, many of them not yet existing. A component should ideally be precisely (formally) specified, easy to understand, sufficiently general, easy to adapt, easy to deliver and deploy, and easy to replace.
- **System development with components** is focused on the identification of reusable entities and relations between them, beginning from the system requirements and from the availability of already existing components [BCK98, GAO95]. Much implementation effort in system development will no longer be necessary but there are efforts required in dealing with components, including locating them, selecting those most appropriate, adapting them, and verifying them [MSP+00].

We not only recognize different activities in the two processes, but also find that many activities can be performed independently. In practice the processes are often already separated, since third parties, independently of system development, develop many components. Even components developed internally within an organization that uses the same components in different products, are often treated as separate entities developed separately. For this reason we can distinguish:

- Life cycle of component-based systems
- Life cycle of components

2.1 Life cycle of Component-based Systems

Development with components differs from traditional development through its focus on the identification of reusable entities and relations between them, starting from the system requirements. Different life cycle models, established in software engineering, can be used in component-based development,but modified to emphasize component-centric activities. Let us consider, for example, the waterfall model using a component-based approach. The top half of Figure 1 shows the phases of the waterfall model. Underneath are shown the accompanying activities in component-based development.

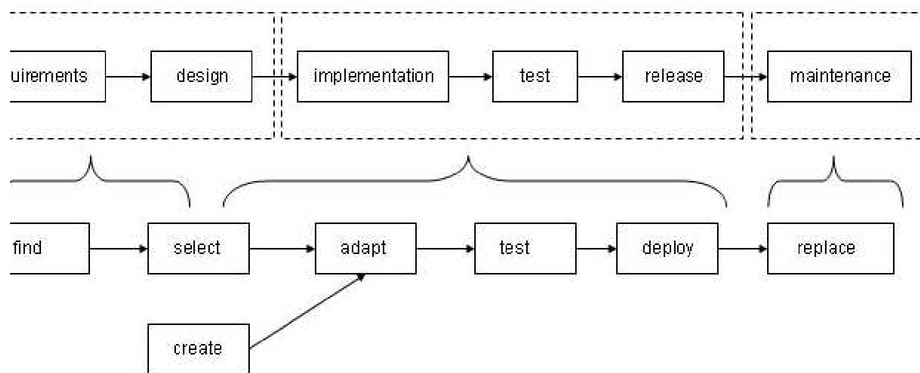

Figure 1. The development cycle compared with the waterfall model.

Characteristic features of component-based development are the following.

- The initial identification of **requirements** is performed as in traditional development. However in the component-based approach, the mapping between system and component requirements is important. Requirements for components should be identified during system requirements elicitation, in order to reuse existing components.
- The early **design** phase focuses on two essential steps:
  - The *logical view* of the system is specified by a system architecture in terms of components and their interaction. In this view, components are represented by their interfaces, possibly including specification of relevant extra-functional properties (in real-time systems this includes timing properties). The architecture specification process is combined with finding, evaluating, selecting, and adapting components that will perform the roles defined by the system architecture.
  - The *structural view* is specified by a system architecture consisting of component implementations, which must conform to a particular component model, assuming a particular system architecture, component framework, and different technology-specific services.
- The **implementation** phase includes adapting, composing, and deploying components, using a framework for components.
- The **verification** (or test) phase performs system verification (e.g., by testing). Rich component interfaces enable a significant part of system verification to be performed in the design phase, thus saving significant effort in the test phase.

- The **maintenance** phase puts extra focus on the replacement and update of entire components, possibly during system operation.

In summary, the activities that are specific to component-based systems development are:

- *Specify logical and structural system architecture*
- *Find and select components that may be used in the system.*
- *Alternately, create proprietary components to be used in the system.*
- *Match component requirements with system requirements (in the specified architecture), and (where possible) verify system properties from component properties*
- *Adapt the selected components so that they suit requirement specification and the system architecture*
- *Compose and deploy the components using a framework for components.*
- *Replace earlier with later versions of components.*

A short overview of these activities is given below:

- *Specify logical and structural system architecture.* The functional and logical system architecture will be the result of an early design, based on system requirements in which design methods, also valid in a non component-based approach, are used. The architecture specification process must take into account that the system requirements should be compatible with those of available components; in this way the system design becomes an interplay to match system and component requirements. Often the requirements cannot be fulfilled completely, and a trade-off analysis is needed to adjust the system architecture and to reformulate the requirements to make it possible to use existing components. In addition, the selection of a particular component technology must be taken into consideration, as a component technology may require particular implementation architecture and includes a number of services such as component intercommunication.
- *Find and select components that may be used in the system.* Available components are collected for further investigation. To successfully perform this procedure, a significant number of possible candidates must be available The selection process is bi-directionally related to the requirements elicitations and system design. If the process begins only with requirements selection, it is highly probable that components meeting all the requirements will not be found. If components are selected too early in the process, the system obtained may not meet all the requirements.

  Component repositories offer tool support for this process. Finding components, testing them in a particular environment and storing them in component databases are activities that can be separated from the system development, but obviously the type of categorization and the search criteria offered by such a repository influences its usability.
- *Alternately, create proprietary components to be used in the system.* In many cases, it will not be possible to define the entire system from already existing components. Especially, the core functionalities of the product are likely to be developed as they should provide the competitive advantage of the product. Parts created in this way should be designed as components with well defined interfaces, so as to allow reuse in a forthcoming application and to facilitate maintenance. This alternative usually requires more effort and lead-time than adapting existing components.
- *Match component requirements with system requirements and verify system properties from component properties.* A research challenge today is to predict the system properties from the component properties. The emerging properties, i.e., the (typically extra-functional) system properties not existing for the components, are of particular interest. For this purpose, *rich interfaces* are essential.

- *Adapt the selected components so that they suit the existing component model or requirement specification.* Some components can be directly integrated into the system, some need to be modified through a parameterization process, some need wrapping code for adaptation, etc. In some cases, it may not be possible to reuse a component itself, but only its interface, which has to be refined and implemented again. This kind of reuse also speeds the development cycle, as at least the interface used in system V&V need not be redeveloped.
- *Compose and deploy the components using a framework for components.* Typically the component framework will provide that functionality. To obtain a particular function, often several components must be composed into an assembly. By introducing assemblies into the system, conflicts between the basic components can occur. It may happen, for example, that assemblies include different versions of the same basic component. In such a case a mechanism for re-configuring assemblies must exist, either supported by the component framework, or used manually. The traditional V&V integration activities must be performed. However, they may become easier if some of the work has been done when specifying the system architecture (predicting system properties from component properties).
- *Replace earlier with later versions of components.* This corresponds to system maintenance. (Implementations of) components, and thus the entire system, may evolve over time. Bugs may have been eliminated or new functionality added.
  - Elimination of bugs in component implementations, which do not affect the interface, should be completely transparent to the system behavior. Ideally, this requires at most a validation of the new implementation against its interface.
  - Any evolution of the system that affects its interface requires an additional validation at system level. If functionality is added, a minimal validation consists in checking that the new functionality is not used in an undesirable manner by other components.

  A particular challenge is to upgrade or replace components during system operation.

2.2 Life Cycle of Components

The component development process is in many respects similar to system development; requirements must be captured, analyzed and defined, the component must be designed, implemented, verified, validated and delivered. When building a new component the developers will reuse other components and will use similar procedures of component evaluation as for system development. There are however some significant differences: Components are built to be part of something else. They are intended for reuse in different products, many of them yet to be designed. The consequences of these observations are the following:

- There is greater difficulty in managing requirements, caused by the interplay between component and system requirements.
- Precise component specifications are more important.
- Greater efforts are needed to develop reusable units,
- Verification against component specification must be more stringent and documented, in particular when transferring components between organizations.
- In a market for components, property rights and their protection become an issue.

The delivery result may be a component, tested and specified, stored in a component library in a package suitable for distribution and deployment. The next phase in the life cycle is component deployment into a system. The deployment should be enabled without making changes in the rest of the system and should happen in an automatic way.

2.3 Issues Specific for Embedded Systems

The design of real-time systems must consider constraints that do not apply to large component- and object-based systems such as business data processing systems. Additional constraints include the following.

- Real-time systems must satisfy constraints on extra-functional properties such as timing (e.g., meeting deadlines), quality of service (e.g., throughput), and dependability (including reliability, safety, and security).
- It is often important that functional and extra-functional properties be statically predictable, in particular if the system is safety-critical.
- Real-time systems must often operate with scarce resources (including processing power, memory, communication bandwidth).

Therefore, definitions and conclusions that hold for large business data processing systems may have to be reconsidered for real-time and embedded systems.

- The most widely used definition of components, by Szyperski [Szy98], emphasizes contractually specified interfaces, fully explicit context dependencies, independent deployment, and third-party composition. It seems biased towards component models where components are deployed at run-time into the system, with run-time support for component registration and composition. This suits well to the component models that are used in non-critical, non-real time, and less resource-constraints applications. However it is not likely that similar properties can be applied to component models for embedded and real-time systems [CL02, Ch. 13]. There is a wide range of embedded systems (from very small to extremely large systems) and there is a wide range of real-time requirements (from hard real-time to soft real-time). While larger embedded systems may afford more resources and in this way provide better prerequisites for utilizing the most widely used component technologies, smaller embedded systems usually cannot afford such resources.
- In widely used component technologies, the interfaces are usually implemented as object interfaces supporting polymorphism by late binding. While late binding allows connecting of components that are completely unaware of each other beside the connecting interface, this flexibility comes with a performance penalty which may be difficult to carry for small embedded systems. Therefore the dynamic component deployment would not be feasible for small embedded systems.

Taking into account all the constraints for real-time and embedded systems, we conclude that there are several reasons to perform component deployment and composition at design time rather than run-time:

- This allows composition tools to generate a monolithic firmware for the device from the component-based design.
- This allows for global optimizations: e.g., in a static component composition known at design-time, connections between components could be translated into direct function calls instead of using dynamic event notifications.
- Design-time composition could be the instance of specific adaptation of components and generated code towards specific micro controller families and real-time operating systems APIs.
- Verification and prediction of system requirements can be done statically from the given component properties.

Design time composition presupposes a *composition environment* that specifically provides the following functionalities.

- Component composition support;
- Component adaptation and code generation for the application;
- Building the system by including selected components and components that are part of the run-time framework;
- Static verification and prediction of system requirements and properties from the given component properties.

There may also be a need for a run-time environment, which supports the component framework by a set of services. The framework enables component intercommunication (those aspects which are not performed at design time), and (where relevant) control of the behavior of the components.

2.4 Summary and Conclusions

Component-based approach faces many challenges. This is in particular true for real-time and embedded systems. The list below emphasize some of them.

- *Component specification:* in the context of embedded systems, it is obvious that interface specifications of components must go beyond syntactic information and include functional and extra-functional characteristics and requirements. For real-time systems the temporal attributes of components and systems are of main interest. For embedded systems the properties specifying the resources and the properties related to dependability are important. However, there is still no consensus about how components for real-time systems should be specified.
- *Component evaluation and verification (possibly for certification):* the trustworthiness of a component, that is the reliability of component in relation to its interface specification is an important issue. The issue is difficult since the trend is to deliver components in binary form and the component development process is outside the control of component users. Protocols for component certification are of great interest.
- *(static) Prediction of system properties from component properties:* Even if we assume that we can specify all the relevant properties of components, it is not necessarily known how they will determine the corresponding properties of systems of which they are composed. Moreover, existing component models do not provide support for predictable composition. In this, one should aim for interfaces providing full functional and extra-functional specifications of components are essential.
- *Component models:* Component models for real-time systems are still in the very early phase of development. In general, existing component models do not support the needs of real-time system development.
- *Architecture specification:* the use of components has an impact on the choice of the system architecture, as it must take into account not only the requirements, but also the available components.
- *Managing the interplay between achievable system requirements and component specifications:* is complex, as the possible candidate components usually lack one or more features which the system requires. Further, the relations between the system requirements and component requirements are complex.
- *Managing changes in component requirements:* an important issue are changes to components over time and possible conflicts arising from different coexisting versions of a component within the same system. A precise interface specification should allow clarifying this issue.

- *Update and replacement of components at run-time* is useful for many real-time systems. In the context of design-time composition, it is a challenge to combine this feature with design-time optimizations across component boundaries.
- *Tool support:* The existence of appropriate tools is essential for a successful component-based development. In non real-time domains there exist various tools supporting a component-based development and they have proved to be successful, but in the real-time domains there is a lack of such tools.
- *Component repositories:* which address the issues of how to store and retrieve components, how to index components in a component library, and how to find "similar" components.

3 Situation and Needs in Different Industry Sectors

The current state of and the needs for component-based approach differ very much between industrial domains. Types of embedded systems vary from ultra small devices with simple functionality, through small systems with sophisticated functions, strong real-time requirements, and low resource consumption requirements, to large, possibly distributed systems, where the management of the complexity is the main challenge. Further we can distinguish between systems produced in large quantities, in which the low production costs are extremely important, and low-volume products in which the system dependability is the most important feature. Usually for high volume products the time-to-market requirements are extremely important as well as the variation of the products. All these different requirements have impact on feasibility, on use, and on approach in component-based development. In different domains we can find very different component models and system and software architectures.

3.1 Automotive

State of the practice

Within the automotive industry, the component-based approach has a relatively long tradition, as these systems are typically built from physical components that are either developed in-house or provided by external suppliers. Today, the physical components also include several computer nodes (or Electronic Control Units, ECUs) equipped with software that implements vehicle functions. A rapid development of electronic components and replacement of mechanical components has increased the importance of efficiency in development and production of embedded components: Modern vehicular systems contain almost hundred computer nodes, and the development costs of the electronic parts have for high end models passed 50 % of the total costs. Even if this development is successful in many aspects, for example in the form of reuse and time-to-market, the trend cannot continue as the systems are becoming too complex and too costly with the current practice of essentially having each ECU dedicated to one function. The current approaches are only beginning to consider the deployment of model-based development processes, and current methods and tool support are available for single ECU based implementations only. The main benefits of the technology used today are improved and more flexible functional behavior of the vehicles, decreased time-to-market and production costs. The trend to replace mechanical components with electronic components will be strengthened.

Current Limitations and the Needs for Improvements

As the number of ECUs increases, the entire system becomes more complex. The system functions, controlling particular aspects at the system level (for example cruise control) require input and output control of many components. This requires sharing different types of resources (time, communication, memory and CPU consumption). With increasing complexity, system reliability and safety become major problems. A satisfactory handling of safety-critical functions, such as

emerging brake- and steer-by-wire systems, will require the integration of methods for establishing functional and temporal correctness for each component, as well as system-wide attributes such as safety and reliability.

Even if the component-based approach is strong on the system level, this is not true for software development. ECUs include proprietary software, mostly owned by subcontractors. This makes the entire system inflexible and inefficient in utilizing resources, makes it difficult to implement complex functions, and expensive to add new ECUs. The next major step in designing these systems is to go from the current situation with "one node – one supplier" to a situation with "one node – several suppliers", i.e., there will be several software components of different origins executing on a typical node. This requires changes in the design process and new division of responsibilities.

A standard or de-facto standard component model for small embedded systems in the automotive domain does not exist today. The existing component-based technologies require too many resources to be suitable for small embedded systems. Developing and establishing an appropriate component technology, including a supporting framework is one of the main research challenges

An example of an ongoing effort in the European automotive industry is the project EAST-EEA (http://www.east-eea.net) with participation of all major European car manufacturers, suppliers and software-tool providers, as well as research organizations and universities with connections to the automotive industry. The goal of EAST-EEA is to develop a structure for the next generation of electronic automotive features. There are two main activities to achieve this goal. (1) Specification of middleware suitable for the automotive industry, and (2) development of an Architecture Description Language (ADL). The middleware specification will leverage on the automotive industry's positive experiences of the RTOS standard OSEK, and will support concepts and provide services on a higher abstraction level than an OS does. The ADL will allow manufacturers and their suppliers to exchange requirements, specifications and documentation about both hardware and software characteristics. The ADL will support system-descriptions on multiple abstraction levels, ranging from very high-level feature specification to very implementation-close operational specifications.

3.2 Industrial Automation

State of the practice

In the last five years the use of component-based technologies has rapidly expanded and become the dominating development technologies in industrial automation. The technology mostly used in large systems is Microsoft COM, and to smaller extent different implementations of CORBA, although neither COM nor CORBA provide support for real-time. The systems using these technologies are soft-real time systems. Often a component technology is used as a basis for additional abstraction level support, which is specified either as standards or proprietary solutions. Some examples of utilization of component technologies:

- **Example 1:** OPC Foundation, an organization that consists of more than 300 member companies worldwide, is responsible for specifications that standardize the communication of acquired process data, alarm and event records, historical data, and batch data to multi-vendor enterprise systems and between production devices. The specification is based on standards DCOM [BK98], XML-DA and SOAP .
- **Example 2:** ABB Automation Products develops a next generation of automation system architecture called Aspect Integrator Platform [CL02, Ch. 17], which is the basis for the design of automation systems, such as open control systems for continuous and batch type processes, traditional supervisory control and data acquisition systems, and others. The architecture uses Microsoft's COM technology, but it determines system architecture and enables flexible system configurations. The main concept is based on AspectObjects which are treated as components. An AspectObject encapsulates all the assets called Aspects belonging to that

objects. In this model the aspects are treated as object attributes. The attributes (as the AspectObject itself) are implemented as special COM objects.

- **Example 3:** Component-based development has been utilized for many years by developing and using the standard IEC 61131 [IEC95]. IEC 61131 defines a family of languages that includes instruction lists, assembly languages, structured text, a high level language similar to Pascal, ladder diagrams, or function block diagrams (FBD). Function blocks can be viewed as components and interfaces between blocks are released by connecting in-ports and out-ports. Function block execution may be periodic or event-driven. IEC 61131 is successfully used in development of industrial process automation systems, for example in ABB and Siemens.

- **Example 4:** Controllers that fulfill real-time requirements (either soft or hard) usually do not use component-based technology such as COM. However in some cases (such as for ABB controllers) a reduced version of COM has been used on a top of a real-time operating system [LCS02]. The reused version includes facilities for component specification using the interface description language of COM, and some basic services at run-time. These services have been implemented internally.

Benefits from Using Component Technologies

The main reason for wide use of component-based technology in the automation industry is the possibility of reusing solutions in different ranges of products, efficient development tools, standardized specifications and interoperation, and integration between different products. For example, the main advantage of OPC is the use of standard interfaces and communication protocols of control devices provided by different vendors. Another benefit is transparency of data access, provided by the middleware. Finally, component-based technologies enable seamless integration with other type of systems, for example business and office applications.

Current Limitations and Needs for Improvements

Many problems originate from the lack of support for real-time and quality-of-service properties in currently used component technologies. Among other things, this results in a need for extensive testing, and in integration problems for large systems.

Because of the dependency on one component technology vendor, there is a standing risk: the current technology can become obsolete and the companies are forced to migrate to new technology even if there are no requirements for that. The controllers, usually hard real-time systems with restricted resources cannot directly use de-facto standard technologies. They either use proprietary component models or try to use particular parts of de-facto standard technologies (for example interface specification, but not run-time support). In the latter case, the challenge is to identify a proper level of reuse of the technology. The lowest level includes use of standardized interface specification, such as IDL (Interface Definition Language), or COM binary interface, and implementation of some standard interfaces.

3.3 Consumer Electronics

State of the Practice

For high-volume electronics products, like TV, VCR, and DVD, cost per product unit is an important issue. These costs are largely determined by the hardware costs, and lead to constraints on the software; for example the available memory. In addition, the diversity of these products increases, as does the complexity of the products due to convergence of functionality. Consumer electronics products are developed and delivered in form of *product families* which are characterized by many similarities and few differences and in form of *product populations* which are sets of products with many similarities but also many differences.

Production is organized into product lines - this allows many variations on a central product definition. A *product line* is a top-down, planned, proactive approach to achieve reuse of software within a family or population of products. It is based on the use of a common architecture and core functions included into the product platform and basic components. The diversity of products is achieved by inclusion of different components.

Because of the requirements for low hardware and production costs, general-purpose component technologies have not been used, but rather more dedicated and simpler proprietary models have been developed. An example of such a component model is the *Koala component model* used at Philips [vO02]. Koala is a component model and an architectural description language to build a large diversity of products from a repository of components. Koala is designed to build consumer products such as televisions, video recorders, CD and DVD players and recorders, and combinations of them. A Koala component is a piece of code that can interact with its environment through explicit interfaces only. The implementation of a Koala component is a directory with a set of C and header files that may use each other in arbitrary ways, but communication with other components is routed only through header files generated by the Koala compiler, based upon the binding between components. As Koala components are delivered in source code, it is possible to statically analyze components and systems built of them.

Benefits from Using Component Technologies

There are two main benefits in a product line development.

- Reuse of already existing components and common architecture.
- Separation of product development from component development.

The first benefit is achieved not only through reuse of the core functionality (which includes the architecture solutions and components that build a core-functionality), but also reuse of particular components in different product families. The second benefit is realized by enabling larger development time for particular components than the time for development of a specific product. Typically, products are released two times per year, while development of a new component requires a year or a year and a half.

There are other benefits resulting from using a component-based approach. The latter forces the software to be explicitly structured. Software components can only interact through well-defined interfaces. The components can be parameterized by the use of diversity interfaces. By binding components into a product, before the actual compilation of the code, the memory footprint can be reduced: optimizations are done across components, without breaking the encapsulation provided by the components.

Current Limitations and Needs for Improvements

The component models used in consumer electronics are proprietary, which requires internal support for their development and maintenance. Further it requires development of a number of development tools: ADL, component repository, composition languages, compilers, debugging and testing tools, configuration tools, etc. Such development is usually not a core business and it requires a lot of resources. Also, the use of a proprietary technology makes it more difficult to use COTS components. There are increasing requirements for achieving interoperability between proprietary and standard component technologies.

The component models used in consumer electronics support only rudimentary analysis and prediction of extra-functional properties of the components and systems. There are increasing requirements for developing methodologies for reasoning about system properties derived from the component properties.

Component models in this domain cover composition at development (compilation) time; run-time systems are monolithic applications, which makes on-line updates of components difficult. Although the requirements for plug-and-play concept are not highly prioritized, it is expected that this will be more important in the future. For this reason a support for managing components at run-time will be required.

3.4 Telecommunication Software Infrastructure

State of the Practice

A main requirement in the telecommunication domain is that service design and development should be done within short durations. But, the infrastructure (middleware) complexity, the heterogeneity of the standards for protocol exchanges, as well as their continuous evolution, the emergence of new ones, and the absence of formal specification for many standards, lead the application designers in the past and still currently to build new applications in a vertical way. Such a vertical structure is contradictory with the need to rapidly build and modify (customize) new services, to integrate them in a consistent way with existing ones, to share common infrastructure and platforms (core network execution platforms or embedded mobile devices), etc. Real-time constraints and QoS requirements are often neglected and consequently, time or performance problems are discovered once the application is deployed. This leads to expensive time to market development and deployment, mainly in the validation and integration phases, and is the source of a crucial telecommunication problem: *service interaction*.

In telecommunication infrastructure, components play and have played a crucial role, and the majority of these components are embedded in core network platforms or several types of devices: mobile, fixed, etc. But, due to the absence of formal specification of component interfaces and composition rules and the lack of real-time and QoS property specifications, current practice consists more in creating new software components (even at the specification level) rather than in reusing existing ones.

Current Limitations and Needs for Improvements

The challenge for the telecommunication domain for the future is to enable the ubiquitous "anything, anytime, anywhere" concept, which means that a service should be seen for an end user as a black box respecting functional and extra-functional properties (Quality of Service) independently of the underlying architecture. Nowadays, due to openness of the telecommunication architecture, a multiplicity of services and service components are provided by several companies and must be dynamically added and updated. Telecomunication applications should be created in a secure and reliable way as fast as possible in a multi-provider environment. In order to achieve this goal, it is essential to provide service designers with software infrastructure offering an interface layer that hides as much as possible the heterogeneity and the complexity of the underlying layers, in order to allow flexible evolution of the applications and of the underlying components, as well as consistent integration of different applications developed by different providers. Concepts such as *Model Driven Architecture* [MM01] have the objective to help the creation of such an infrastructure. It is time to go from ad-hoc techniques for component composition towards more integrated and formal ones. Components from different horizons are omnipresent in telecommunication and they have to be integrated in a consistent way. There is a real need to go from a vertical service development to a horizontal approach based on flexible, reliable and open software infrastructure (middleware).

A research goal for software infrastructure should be to integrate Model Driven Architecture concepts with component based development approaches. We need also an innovative and consistent development methodology from high level specifications towards design. Formal validation of components, which must respect a rich interface specification, is crucial for a consistent composition of distributed components. It is the only way to ensure a flexible and secure interface to the telecommunication service designer.

For the next generation telecommunication software infrastructure, several aspects of components are important:

- Specific attention should be paid to mobile devices. They have to tackle several critical constraints (memory size, energy consumption, time constraints, etc.). They require continuous adding, removing or modification of components, and different service negotiation procedures. Security and availability should be provided in any kind of environment (unreliable environment, different kinds of communication modes, different performance properties). Specific components should be provided for different communication patterns.
- In order to cope with the problem of interoperability between different protocols and API implementations, standards are defined, which specify syntactic interfaces as well as service requirements, timing aspects and other extra-functional assumptions on properties of the underlying network. Standards are often given by documents in natural language (or in some cases by UML class diagrams) and focus on static specification, but their ambiguity leads to inconsistencies in dynamic and real-time behaviors. It implies the need of an increased use of modeling languages with a formal executable semantics, like SDL, which allow to fully specify the expected interface behavior.
- For service designers, the interest of components goes beyond interoperability. Service components have individual requirements that might be violated when composed and deployed with other service components. This problem, well-known in the telecommunication world as the service interaction problem, should be tackled taking into consideration real-time and performance aspects. Especially, in the context of mobile telecommunication or WEB-services, real-time aspects, quality of service and dynamic composition are important issues.
- Another important aspect is the definition of a methodology for component based design, from the analysis steps towards implementation and testing, applied for component life cycle and system life cycle. There is a large consensus for the use of standards such as UML, SDL and MDA in the telecommunication world, but research is necessary in order to take into account real-time aspects, quality of service and deployment issues and to better integrate components and composition in the software life cycle.
- Component and system verification using formal techniques for real-time systems should be enforced. It should enable quick and secure telecommunication service creation answering questions like how to build an architecture based on a set of components (reused and/or shared by several services) in such a way that we can guarantee the provision of complete applications respecting quality of services and safety requirements (especially security requirements).

3.5 Avionics and Aerospace

Characteristic for software development for avionics and aerospace include the following.

- Application are highly safety- and mission-critical.
- Systems are inherently complex and expensive to design, upgrade, and support.
- Systems have an extremely long lifetime and will undergo several generations of upgrades and platform migrations.
- Extensive simulation and V&V are performed, since flight testing is extremely costly.

One consequence is that model-based approaches are more advanced and applied than in other domains, e.g., as witnessed by the prominence of the avionics application domain in many advanced technology projects (e.g., SafeAir http://www.safeair.org/project/, Mobies http://www.liacs.nl/ marcello/mobij.html, and others).

Perceived Benefits from Using Component Technologies.

Particular attention has been paid to modeling of system and software architecture, and to the development of supporting technologies (code generation, middleware). Benefits from component technology include the following.

- Support for model-based development,
- Platform independence is achieved by supporting middleware and mapping tools.
- System evolution is supported by ability to replace individual components, perhaps in the context of a developed product line architecture.

An example of technologies that have been developed for the avionics domain is MetaH. MetaH is a domain-specific ADL dedicated to avionics systems which has been developed at Honeywell Labs since 1993 under the sponsorship of DARPA and the US Army. A significant set of tools (graphical editor, typing, safety, reliability, and timing/loading/schedulability analyzers, code generator...) has already been prototyped and used in the context of several experimentation projects.

Current Limitations and Needs for Improvements

Standardized component models seem not to be widely used, maybe because a large portion of software development is performed in-house with rather specific requirements. More emphasis appears to be placed on predictability of global system properties and global system architecture.

A prominent desire is to continue the trend towards model-based development, supporting it by integrated tool chains that can perform analysis of properties like fault tolerance, timing, utilization, quality of service, etc. on models, and thereafter generate optimized code for target platforms. There should be infrastructure support for distribution.

3.6 Summary and Conclusion

Components have been used in several industrial domains. The component-based approach on system level, where hardware components are designed with embedded software, has been successfully used for many years. Also large-grain generic components like protocol stacks, RTOSs, etc. have been used for a long time. In addition to this, technology supporting a component-based approach has been developed, either in the form of proprietary component models, or by using reduced versions of some widely used component models.

Main benefits include the following.

- Achieving time-to-market requirements by separating component development process from system development process.
- In some sectors, reuse has been supported by the creation of product-line architectures.
- Imposing structure on complex systems, thus mastering complexity and making maintenance easier.
- Solving problems with scalability, including integration of large systems on platforms with limited resources. Some component-models use components with parameterized interface that makes it possible to define the amount of memory and other resources used.

It should be noted that the following needs are important to different degrees in different sectors.

- The adoption of component technology is hampered by the lack of widely adopted component technology standards which are suitable for real-time systems.

- There is also a need to ensure interoperability between different component technologies. One motivation is for users not to be bound to a single vendor of platforms or integration tools.
- Specification technology is not sufficiently developed to guarantee a priori component interoperability. This is important, e.g., in the telecommunications domain where interoperability is crucial, and in domains where manufacturers have the role of system integrators.
- Reuse of components across different organizations is hampered by the lack of technology and procedures for verifying and certifying component implementations.
- Most current component technologies do not support important extra-functional properties, such as timing behavior or QoS properties, that will be needed for predictability of embedded systems. Explicit management of timing requirements is necessary when building component-based real-time systems. A current shortcoming is that methods for breaking down system timing requirements into component requirements are not fully developed.
- There is a need for generic platform services, for, e.g., security and availability.
- Tools that support component based development are still lacking. For proprietary component models, the cost of developing such tools can in general not be motivated. The lack of tools to enforce conformance to interfaces and standards puts heavy burdens on component and system developers and limits adoption of component technology.
- For smaller-size embedded systems, it is important that a system composed of components can be optimized for speed and memory consumption, typically by globally optimizing compilation. This applies to sectors with large volumes and small platforms that have constraints on, e.g., power consumption, such as the automotive industry and small mobile devices.
- There is a trend towards open, extensible, and upgradable systems. This creates a need for systems where components can be deployed or upgraded at run-time. This requirement conflicts to some extent with the desire for compile-time global optimizations, implying that domain- or application-specific trade-offs may be needed.
- To support more advanced component technologies for embedded systems, it is important to develop efficient implementations of component frameworks (i.e., middleware), which have low requirements on memory and processing power.
- Components have individual requirements that can be violated when composed and deployed with other components. Techniqes are needed that ensure that components do not interfer with requirements of other components. Such interferences can be obvious, such as violations of memory protection, or more subtle. An important scenario where interferences will occur is when several components, each implementing a piece of functionality, are mapped onto one ECU.
- There is a desire to develop domain-specific architectures, which can define different roles for different types of components.

4 Specifying and Reasoning about Contracts

4.1 Introduction

One of the key desiderata in component-based development for embedded systems is the ability to capture functional and extra-functional properties in component interfaces, and to verify and predict corresponding system properties. For real-time systems, this is perceived to be particularly important for properties such as timing and quality-of-service.

In this section, we review existing techniques for capturing, verifying, and predicting different properties of component and system behavior. Properties of components can be expressed in

their *contracts*, hence the title of the section. To structure the exposition into different types of component properties, we use the classification of contracts proposed by Beugnard et al. [BJP99], where a *contract hierarchy* is defined consisting of four levels.

- **Level 1:** Syntactic interface, or *signature* (i.e. types, fields, methods, signals, ports etc., that constitute the interface).
- **Level 2:** Constraints on values of parameters and of persistent state variables, expressed, e.g., by pre- and post-conditions and invariants.
- **Level 3:** Synchronization between different services and method calls (e.g., expressed as constraints on their temporal ordering).
- **Level 4:** Extra-functional properties (in particular real-time attributes, performance, QoS (i.e. constraints on response times, throughput, etc.).

Currently, most component models support only level 1 contracts, while some models support also other levels (see Section 5). In the remainder of Section 4, we will survey techniques for capturing and reasoning about component and system properties, discussing each aspect separately. We will use the four levels of the Beugnard hierarchy for structuring our treatment of different interface properties. Regarding level 4, we make a separation between timing properties (e.g. absolute time bounds) and stochastically formulated performance properties (e.g. average response time). In addition, we briefly treat reliability properties.

For each aspect, we will consider techniques for

- expressing properties of systems and components,
- predicting or verifying system properties from component properties, in particular for doing this statically at design-time,
- checking that component properties are compatible (assumptions made in one component specification are guaranteed by some other component specification),
- verifying that component implementations satisfy properties given in component specifications,
- and compile-time and run-time support for enforcing system or component properties.

On the Nature of Contracts

The term *contract* can very generally be taken to mean "component specification" in any form. A contract is in practice taken to be a constraint on a given *aspect* of the interaction between a component that supplies a service, and a component that consumes this service. Component contracts differ from object contracts in the sense that to supply a service, a component often explicitly requires some other service, with its own contract, from another component. So the expression of a contract on a component-provided interface might depend on another contract from one of the component-required interfaces. For instance, the throughput of component A doing some kind of computation on a data stream provided by component B clearly depends on the throughput of B.

It is indeed challenging to develop a practical framework for reasoning about complex component properties (e.g., performance properties) stated in contracts, e.g., to infer global system (performance) properties. A complete solution to this problem requires powerful mathematical reasoning, e.g., about properties of stochastic processes. A pragmatic, more modest, approach to this problem, which does not need powerful mathematical reasoning, is to agree on a small set of fixed contracts, or a small set of fixed building blocks for contracts. For each contract, one can then in advance develop techniques for monitoring or verifying that component implementations satisfy the contract, and techniques for inferring system properties from component contracts. For

instance, for performance properties, one can define a fixed set of different levels of performance, and for each level define rules for run-time monitoring and for component interoperability.

In simple cases, such a scheme can be seen as constructing a type system for specifying properties. More complex cases may involve constraints expressed in some type of logic, and thus checking beforehand that components interact correctly then need some form of theorem proving techniques.

4.2 Level 1 - Syntactic Interfaces

Definition

By a *syntactic interface*, we understand here a list of operations or ports, including their signatures (the types of allowed inputs and outputs), by means of which communication with a component is performed.

Generally speaking, a *type* can be understood as a set of values on which a related set of operations can be performed successfully. Belonging to a given type usually implies constraints that go beyond what value is denoted exactly, most notably *how* the value is stored (required when operations are performed). Once types have been defined, it is possible to use them in specifications of the form: if some input of type t_{in} is given, then the output will have type t_{out}.

Type safety is the guarantee that no run-time error will result from the application of some operation to the wrong object or value. A *type system* is a set of rules for checking type safety (a process usually called *type checking* since it is often required that enough information about the typing assumptions has been given explicitly by the designer or programmer, so that type checking becomes mostly a large bookkeeping process).

"Static" type checking is performed at compile- (or bind-) time and ensures once and for all that there is no possibility of interaction errors (of the kind addressed by the type system). Not all errors can be addressed by type systems, especially since one usually requires that type checking is easy; e.g., with static type checking it is difficult to rule out in advance all risks of division-by-zero errors.

Type systems allow checking *substitutability* when components are combined: by comparing the data types in a component's interface, and the data types desired by its environment client, one can predict whether an interaction error is possible (e.g. producing a run-time error such as "Method not understood").

Specification of System and Component Signatures

A system for specification of syntactic interfaces must include:

- A type system, together with a syntax (we can call it an Interface Description Language, or IDL) for specifying signatures of operations/ports;
- A mapping from the (abstract) interface types to component implementations. For instance, if components are given in some programming language (for example, if they are written in C), and the interface types use the type system of C, then the mapping is direct. If components are available in binary form, there must be an agreed mapping from interface types to binary formats of component implementations.
- A notion of substitutability, which describes when the interfaces of two components are compatible.

For embedded systems, the type system is usually rather simple, with a substitutability amounting to equality (i.e. one may only substitute objects whose interface is the same as the declared one). For run-time component frameworks, a little bit more flexibility is usually allowed, with substitutability based on type extension or even a more generally defined conformance relation.

For instance, every CORBA object has a type name, which is the same as the interface name assigned in its IDL declaration. The operations that it can perform, and the variables (and their types) that it understands, are all part of its type. Base types include three different precisions of integers and floating-point numbers plus fixed-point, standard and wide characters and strings, etc. Constructed types include records ("struct"s), unions, and enumerations. One can declare either fixed or variable length structs, arrays, strings, and wstrings. There is an *any* type that can assume any legal IDL type at runtime.

CORBA supports subtyping by extension: one can create a subtype by extending the base type's list of operation signatures. But one must not redefine any of the base type's operations, and it only works in the absence of explicit self-reference. The advantage of this scheme is that it is easy to implement and understand, the disadvantage is that it is still quite restrictive since some safe substitutions are ruled out.

A proposal for a polymorphic type system suitable for embedded system design is given by Lee and Xiong [LX01] and incorporated in Ptolemy II. It combines several types of polymorphism, including some standard coercions between numeric data types. One design goal is that the check for substitutability should be efficient, since one may have to carry it out at run-time.

Component Interoperability

Conformance is more generally defined as the weakest (i.e., least restrictive) substitutability relation that guarantees type safety. Necessary conditions (applying recursively) are that a caller must not invoke any operation not supported by the service, and the service must not return any exception not handled by the caller. Conformance has a property called *contravariance*: the types of the input parameters of a service must conform in opposite to the types of its result parameters.

For example, if we have a type `sign` for the set of the three numbers -1, 0 and +1, it is natural to see `sign` as a subtype of `integer`. Now consider a numerical function `sign` from integers to signs: this function can be used (substituted) in contexts where a function accepting sign is expected, and in contexts where a function returning integers is expected.

At first, the contravariant rule seems theoretically appealing. However, it is less natural than covariance (where parameter types conform in the same direction), often encountered in real-world modeling (animals eat food, herbivores are subtypes of animals, but they eat grass which is a subtype of food, not a supertype!), and is indeed the source of many problems. The most surprising one appears with operations combining two arguments, such as comparisons. If the contravariant rule is used, the type associated with *equal* for *Child* instances is not a subtype of the one of *equal* for *Parent* instances. As soon as this kind of feature is considered (and they are common), the contravariant rule prevents a subtyping relation between *Child* and *Parent* (see [Cas95] for more details and solutions).

Trends and Conclusion

About 10 years after the debates on contravariance vs. covariance have peaked in the OO research community, the dust has settled down somewhat. We can now identify three main directions that have been taken to deal with this issue.

- (i) Keep it simple: No-variance is used for IN parameters. That is the approach used in mainstream languages such as CORBA, C++, Java, C♯ etc. For instance, if one needs a specialized version of *x.equal(y),* the type checking (through downcasting on parameter *x*) must be done by hand by the programmer, and verified at runtime only.
- (ii) Model reality: Covariance is used for IN parameters. This is the approach used in Eiffel, which makes static type checking a non-local, non-incremental task. Indeed, if no other restriction is made, type checking requires extensive program analysis and looks much more like theorem proving than the simple bookkeeping process it used to be.

- (iii) Make it complex: use parametric polymorphism in conjunction with reference polymorphism, and have a type system where the types themselves can be seen as variables. This is quite appealing as far as the expressive power of the type system is concerned, but it still lacks a mainstream adoption.

The conclusion is that as soon as one wants a minimum of flexibility for defining type conformance between a provided interface and a required interface, static type checking is no longer a simple bookkeeping process. So level 1 contracts do not have a very different nature than contracts of other levels. In some cases, they can be defined with restrictive rules to allow simple tools to process them, in other cases one could be interested in having more flexibility at the price of more complex tools for static checking, or even rely on runtime monitoring.

A concern in component-based design of embedded systems is that runtime monitoring of interface types may be desirable for building reliable systems, and because one cannot completely trust component implementations. If components are deployed at run-time, the check for substitutability must be performed with available computing resources.

4.3 Functional Properties (Level 2)

Definition

Functional properties are used to achieve more than just interoperability. Level 2 in the Contract Hierarchy is concerned with the actual *values* of data that are passed between components through the interfaces, whose syntax is specified at Level 1 (the preceding section). Typical properties of interest are constraints on their ranges, or on the relation between the parameters of a method call and its return value. It is also customary to include at level 2 properties of a persistent state of a component. In level 2 contracts, transactions are described as atomic, which means they are appropriate for components with sequentialized or totally independent interactions.

Specification of System and Component Properties

Formalisms at level 2 provide means for describing partial functions or relations for representing a component (or system) step. In constraint languages, as provided by Eiffel/SCOOP [Mey91, Mey97] (dedicated to the Eiffel programming language), OCL [WK98] (Object Constraint Language dedicated to UML), LSL (Larch Shared Language) [GHG+93], JML (Java Modeling Language) [LB99], relations are expressed by means of *invariants, pre-* and *postconditions.* More classical notations are for example Kahn networks [Kah74]. Logical formalisms are Unity [CM88] or TLA [Lam94], with the difference that they allow also to express liveness properties, that is, additional properties of infinite sequences of steps (fix points).

In practice, pre- and postconditions are rarely used in the context of large components, but rather for small components, often describing data structures providing a set of operations considered as atomic. One reason may be that the same type of interfaces is much harder to obtain for compositions of components.

Verifying Component Properties

There exists a number of tools using constraints for *run-time monitoring* which generate exceptions in case of violation of interfaces at run-time. This is the case for example in Eiffel and for JML annotations of Java. It also exists in dotNET. Run-time monitoring assumes that interface specifications are executable, and incurs a nontrivial cost. Many frameworks use assertions in a test phase, often using a constraint language. Here, aspect-oriented programming techniques [KLM+97], which allow to compose different features when generating code for testing or for final implementation, can be used to introduce some degree of automation and to facilitate maintenance.

There are research tools that perform static checking of JML, such as ESC Java at Compaq [ESC] based on (partial) static analysis methods, or in the Loop project at University of Nijmegen [Loo] which is based on the use of interactive theorem provers. Theorem provers are also used to verify invariants or temporal logic properties on TLA or Unity specifications. Such tools are primarily used in applications that require highly dependable software. Even in the future, they might not become widely used in standard component based development, but it is important that they exist for demanding applications. Certainly, component manufacturers may want to use them, to provide highly dependable component implementations conforming to contracts. To some extent, the use of theorem provers can be seen as a form of experimentation, which should result in automated procedures for various application domains.

Some of these formalisms are also used in the domain of hardware or on finite state abstractions of components, where (symbolic) composition and model-checking are applicable, and any of the many model-checkers developed in the last 2 decades can be used.

The B-Method [Abr96], is based on a formalism of the same kind, but it provides an integrated framework for systematic refinement from invariants to implementations of functional components.

Component Interoperability and System Properties

There are two aspects of interoperability: one is preservation of component properties and general system properties like absence of deadlock, and the second is verification of emerging global system properties corresponding to functional system requirements.

In the context of level 2 specifications, composition of interfaces can be seen as composition of partial relations. Therefore, *component interoperability* amounts to verify that composition does not require strengthening of preconditions (leading to additional undefinedness). In simple cases, it can be sufficient to check that pre-conditions are satisfied by corresponding post-conditions of connected interfaces.

The level 2 *system properties* are determined from the composed partial relation. In general, its formal calculation requires more sophisticated mathematical machinery in the form of fixed-point theory, as simpler representations in terms of invariants and pre/postconditions cannot always be synthesized.

The situation concerning existing tool support is the same as for the verification of components themselves. Run-time monitoring is the main approach, and alternative methods consist in using interactive theorem provers, or, alternatively, a top-down approach based on systematic refinement. In the case of finite domains, the situation is more favorable, as representations of compositions can be synthesized from representations of components.

Research Challenges

Existing academic tools for the static validation of component properties should be pushed towards more automation and integrated into professional development tools. The main problem of level 2 specifications is their applicability to distributed systems, due to the absence of means to express interactions as non atomic or to express explicit concurreny. This can be improved by considering additional level 3 specifications. In practice, level 2 specifications can be used mainly for a single level of components and when non-interference between transactions can be guaranteed by construction (in general by sequentializing access to components), as for example in the synchronous approach used in the context of safety critical applications.

4.4 Functional Properties (Level 3)

Definition

Level 3 in the Contract Hierarchy is concerned with the actual ordering between different interactions at the component interfaces and more importantly, they allow interactions between a component and its environment to be considered non atomic. Level 3 specifications provide the following facilities:

- description of transactions (input/output behaviors) not necessarily as atomic steps.
- explicit composition operators avoid the obligation to provide an explicit input/output relation taking into account all potential internal interactions. This has the further advantage that a restricted use of a component does indeed allow to derive stronger properties (only the actually occurring interactions need to be taken into account, not all hypothetical ones)
- many level 3 formalism allow to express explicit control information, which makes the expression of complex, history dependent input/output relations much easier

Indeed, formalisms at level 3 have explicit composition and communication primitives.

Specification of System and Component Properties

There are several, formally comparable families of description techniques:

- Automata, including hierarchical state machines etc. as found in SDL, UML have explicit composition operators which allow easily to represent complex components by means of the same formalism.
- Process algebras are very similar in principle, and really focus on the notion of composition.
- Temporal Logics are used for the description of global properties to be verified on a component or a system (also for components specified with level 2 interfaces), rarely as component characterizations to be used in further composition.
- Sequence Diagrams or Sequence Charts represent also global properties, but in terms of a set of interesting scenarios and are mostly used to describe test cases. They may be used to describe complete specification if the number of alternative scenarios describing a transaction is relatively small.

In order to distinguish between or the *required* and *offered* parts in the context of contract specifications, most of these formalisms use the distinction between inputs and outputs. Timeouts or explicit timing restrictions can be used in some formalisms to restrict waiting for particular inputs or component reaction time. Most of these formalisms must handle unexpected inputs explicitly by providing complete specifications. A recent suggestion for extending automata-based formalisms with explicit distinction of provided and required interfaces are *interface automata* [CdAH+02].

Verifying Properties and Component Interoperability

In the context of level 3 contracts, the expression of interfaces of complex components is made possible due to explicit composition. In this case, the verification of component properties and of system properties are of the same nature. However, system verification can easily become intractable for systems consisting many complex components (cf. the state explosion problem).

In academia, in the last two decades a large number of model-checking tools have been developed, which allow to show that a composition of automata (describing behaviors of components) satisfies

- that is implements - some property (a desired component or system property), described either as an automaton, a formula of temporal logic, or in the form of a scenario (Message Sequence Chart [Mau96, IT00] or Live Sequence Chart [DH01]). These tools can be used to verify properties of relatively small descriptions, i.e., mainly of medium-size components or systems. In order to make the verification of complex systems that are compositions of components tractable, two kinds of methods have been developed:

- abstraction, to hide the internal structure of sub-components and to synthesize the externally visible behavior of a component by abstracting, whenever possible from interactions between internal components
- compositional verification techniques, which are similar in nature but based on characterizations of components in terms of (temporal) properties and use of deductive verification techniques

Most model-checkers work on finite-state systems only, but in the last years also tools for checking decidable or semi-decidable properties of infinite-state systems (such as parameterized systems, systems with counters or communication through lossy channels) have been developed. Nevertheless, at present these tools are not integrated with any existing tool for component-based development.

For modeling languages like SDL or UML, which can be used to describe interface behaviors, there exist case tools [USE01, Ilo, Tel] with restricted simulation and validation facilities, allowing to validate a composition of a set of components described by their interface behavior by simulation. Nevertheless, none of these tools provides facilities for defining observation *criteria* that are necessary to explicitly hide internal information. Industrial practice is mainly based on testing and/or on model-based simulation. The step from a complete functional model to an implementation, for example in C can be done automatically in some contexts. For synchronous languages, and for SDL, automatic code generators exist and are being used.

It should be noted that system validation is in general not done for arbitrary environments, but with a particular, restricted environment (including the underlying platform) and a restricted number of possible interaction scenarios in mind. This reduces the amount of non-determinism and makes validation more feasible, which then gets very close to testing of a restricted number of scenarios played by the environment. In this context, level 3 specification have the considerable advantage over level 2 specifications that encapsulation of internal activities need not be done a priori for all uses of a components, but after restriction to a particular environment. This allows for the derivation of stronger global properties.

Research Challenges

The short-term perspective is to integrate existing academic model-checking tools to validate component properties, and to derive system properties from (relatively) complete component properties using professional development tools. Another important direction is to develop tools for *run-time monitoring* of properties given in terms of sequence charts, automata, TL, etc.

Compositional methods that allow the derivation of properties of realistically complex systems do not really exist. For this, more research on automatic abstraction and property extraction by static analysis is needed. Nevertheless, there will be no ultimate verification method for the verification of arbitrarily complex systems. When verification is an issue, systems must be built having verification in mind. Requiring only level 2 specifications, which capture only input/output relations, is too drastic. Nevertheless, the identification of adequate restrictions to obtain reasonable complexity at each level of composition is a key issue.

Ongoing interesting work on composition principles that tries to formulate sufficient conditions for guaranteeing the preservation of component properties during composition can be found in [BGS00, GS02]. Principles that allow the inference of system properties directly from component properties, would certainly provide more motivation for the verification of components. An interesting research challenge is the study of architectures that support such composition principles.

4.5 Timing Properties (Level 4)

Definition

Timing requirements define constraints on the order of occurrence and on upper and/or lower bounds of durations between *events*. We can distinguish between *hard real-time systems*, where all the occurrences of the specified events must satisfy the specified constraints, and *soft real-time systems* where the distribution of the durations between the specified events over all occurrences within an execution must obey some constraints, e.g., on average and variance etc. In this section, we consider timing properties for hard real-time systems, for soft real-time and QoS we refer to the next section.

Specifying timing requirements

In **current practice**, time bounds can be associated with the duration between events in an informal or (semi)formal requirements specification. Typical timing properties are the following ones, where time requirements are expressed using physical time, e.g., seconds, some abstract time unit, cycles of some clock or number of computation steps. When different requirements and definitions of a system are expressed using different notions of time, it is important that the relationship between these different notions is well defined.

- When called, this method is computed within 20 ms (execution time property).
- This function is computed periodically, with a period of 50 ms (periodicity property)
- packets are sent with a frequency of 50Hz and a maximal jitter of 1ms (periodicity property)
- Component C receives data requests at most every 3 ms (interarrival time property)
- When the value of variable x exceeds 100, component C is notified within less than 10 ms (reactivity property)
- If lightning strikes, transformers is shut off within 50 microseconds (response time property)
- RPM does not exceed 50000 for more than a few seconds during the start phase.
- The response to this signal comes within 3 cycles (response time property)
- When component C gets a request every 2 to 3 cycles, it provides the response within 2 cycles (conditional response time)
- the execution time of task T is 20 to 30 ms and its overall duration should not exceed 100m.

Note that such properties can express both requirements and assumptions depending how they are used.

Existing **formalisms** that allow the expression of time bounds are in fact extensions of level 2 and level 3 formalisms extended with time. They include metric temporal logics, i.e., temporal logics with quantitative constraints on the duration between events, timed extensions of automata, sequence charts extended with time, timing diagrams or general constraint languages, like OCL, extended with time. For example, in Message Sequence Charts [IT00] and also in Sequence Diagrams in some UML tools, time bounds can be assigned to the distance between two events. In Live Sequence Charts [DH01], time dependent properties are expressed with timers, meaning that durations cannot be measured, but only constrained. Timed automata are more expressive: they specify constraints between events by means of so called "clocks" measuring durations, which are reset to zero at the occurrence of one event, and then used in "guards" to restrict the possible occurrence times of other events. A notion of *urgency* allows to distinguish between time constraints and time guards.

Temporal logics extended with time have rather limited expressive power. Sequence Diagrams define time constraints in the context of certain scenarios, and are very cumbersome if the overall number of scenarios is big. Timed-automata based formalisms naturally define constraints on all possible scenarios, but it is harder to argue about particular "interesting" scenarios.

Some programming and modeling languages have an explicit notion of time. In synchronous languages one can define behaviors occurring at certain cycles, where cycles of various lengths (all multiples of a basic cycle) can exist. In the modeling languages SDL and Room, a notion of global time and timers can be used. But all these formalisms are aimed at the definition of time-dependent behaviors, rather than at expressing real-time requirements. ITU recommends time-extended Message Sequence Charts for defining real-time requirements for SDL system models.

For UML, which contains both formalisms for functional behavior descriptions and for expressing requirements and constraints, recently a "profile for real-time, scheduling and performance" has been defined [OMG01b], including notions of timers, timed events, constraints on their time of occurrence and a large number of notations for which no semantics are given. These notions are defined for all of UML, but have apparently been built with mainly timed Sequence Diagrams in mind. A more elaborated RT-profile for UML, also based on a large number of intuitive notations, but including semantics, is being developed in the OMEGA IST project (http://www-omega.imag.fr/).

Note that component timing properties depend in general on a given platform and system configuration. They must therefore either be properties of a system component (i.e., the running software together with platform and run-time system) be parameterized by (characteristics of) the undelying platform, compiler, etc.

Component Interoperability and System Properties

There is a vast literature on timing analysis, treating the problem of determining whether a set of given system timing requirements can be met by a collection of components with known timing parameters. The most common paradigm is schedulability analysis, which takes as input component timing properties, system timing requirements (on response times, periods, deadlines, etc.), and properties of the scheduler and platform. The output is an answer about feasibility and information about how the scheduling should be performed. In the context of hard real-time systems, it is important to answer the following questions: "to which extent a component based approach is possible?" and what kind of "components" are useful in this context. For this purpose, let us look at what is current practice:

- **Scheduling of periodic tasks.** Mainstream schedulability analysis assume that *tasks* are executed periodically or aperiodically with known maximal activation frequency. For each task a worst case execution time is known or assumed, and where applicable also the worst case communication requirements, overhead for context-switching, etc. on a given platform. A simple framework is RMA, where all tasks are periodic, can be preempted, and have a statically known pattern of access to shared resources. Under suitable conditions, schedulability can be analyzed in a time proportional to the number of tasks. This approach is present in MetaH and Rubus and to some extent in PECOS. In general, an integration platform need not perform the schedulability analysis itself; this can be done by an external tool. Schedulability analysis can also be performed for distributed platforms, if communication delays have known bounds. An example is the Volcano system on CAN. This approach has also inspired the Real-Time profile of CORBA, and in the area of languages, Java-RT and Posix. The approach is mature and has proven practicality.

  In this context, a component may realize tasks or represent a shared "resource" used for the realization of certain sub-tasks. Its interface must, therefore, specify its worst-case execution time for each task (or sub-task) for the platform under consideration and the implied resource usage. It must also be stated whether pre-emption is allowed, and whether multiple concurrent invocations are perimtted.

- **Synchronous Approach.** This paradigm enforces a very strict scheduling policy. Globally,

the system is seen as a sequential system that computes in each *step* or *cycle* a global output to a global input. The effect of a step is defined by a number of transformation rules. Scheduling is done statically by compiling the sets of rules into a sequential program implementing these rules and executing them in some statically defined order. A uniform timing bound for the execution of global steps is assumed (system requirement).

In this context, components are often "design-level components", as the component-based design is compiled into a single sequential program later on. In this case the analysis of the WCET (worst case execution time) of a single step is done on the target code directly. An extension to the use of run-time components consists of generating code containing *calls* to those components. Some component models, such as IEC61131-3 use this execution paradigm. In some sense, this approach is quite close to RMA, where the "global period" plays the same role as the "global step".

TTA defines a protocol for extending the synchronous languages paradigm to distributed platforms. In this context, distributed components can be made easily interoperable as long as they conform to the timing requirements imposed by the protocol.

Another component view consists in considering an entire synchronous system as a "component" communicating (asynchronously) with its environment by buffering inputs from the environment and/or relying on certain continuity properties of the environment. This is sometimes called the GALS (Globally Asynchronous, Locally Synchronous) approach.

- **Generalizations:** Currently under investigation in the research community are generalizations of schedulability analysis to distributed systems and to more dynamic task sets, e.g. with reconfiguration (this is discussed in action 3). Another extension consists in considering components with a more complex structure than entities realizing a set of periodic tasks with a global WCET, e.g., components which have an internal state, as described by a state machine, or systems with modes. Quite a number of tools have been developed recently, aiming at analysis of this kind of systems, such as Taxys [BCP+01], Prometheus [Gös01], IF [BGM02], or Times [AFM+02] . An ambitious example of a model and framework supporting this and other paradigms is Ptolemy [RNHL99, Lee01] or Metropolis [BLP+02].

In this context component interoperability is to some extent subsumed under the timing analysis done when checking that system requirements can be met. This analysis includes checking that the components can cooperate to satisfy the system timing requirements. In the context of soft-real-time system, composition frameworks as proposed in [BGS00] are very promising.

Verifying Component Properties

A difficult point in timing analysis is assessment of WCETs of tasks or of the code implementing a global step of a synchronous system. In current practice, this is done by measurements (on each particular target platform), or by simulation, e.g., by using hardware simulators. Recently, there has been progress in static code-based prediction of WCET by taking into account a very precise platform model [FHL+01, FW98]. Note that WCET calculation is becoming more and more complex, since new hardware features of processors are increasingly unpredictable, and due to the sometimes complex platform dependencies. In order to make assertions about upper bounds of durations, both time-dependent characteristics of the external environment and of the platform on which the component is executed, as well as knowledge about all resource usage, need to be known. Work on extracting timing information from peripherals and other devices remains to be done.

Research Challenges

Recently, important advances have been made in the domain of timing analysis, as well concerning execution time estimation for individual tasks, and also concerning interoperability and scheduling analysis. Design-time timing analysis is being integrated into component-based design.

Often, traditional scheduling theory is adopted: parameters of some scheduling paradigm are added to the interface/specification of components. Such interface/specifications are also standardized [OMG01b]. Classical scheduling theory assumes that system architectures and components have a certain structure. A number of approaches exist which go beyond the classical theories and propose techniques to extend timing analysis to less constrained forms of component specifications (e.g., as timed automata). Nevertheless, a number of open and challenging problems remain to be solved:

- The major effort in the **short term** is to integrate the analysis capacities of the above-mentioned tools for both
  - low-level timing analysis based on abstract interpretation and
  - for interoperability analysis based on timed automata or other more general task descriptions into component-based design.
- Presently, there exist a number of approaches trying to extend the well-understood, but restrictive paradigms provided by the synchronous approach or by classical RMA analysis to more general frameworks. For example, TTA is an approach extending the applicability of the synchronous approach to distributed systems. A system like Giotto[Hen01] extends it with more dynamic scheduling of the tasks making up a step.

  A really challenging research task consists in the development of a paradigm that encompasses the whole spectrum of approaches from the very strict synchronous approach to the fully asynchronous approach, including distributed systems. Such a paradigm must provide a semantic framework for composition of time-dependent components, based on different communication and interaction modes. This will allow the verification of compositions of time-dependent systems and their properties at modeling level.
- A number of features, such as run-time update and dynamic reconfiguration of systems, which provide some of the motivation for using a component-based approach, have so far been essentially avoided in systems with hard real-time requirements. It is an interesting research question, whether such features can be reasonably included into hard real-time systems.

We anticipate that componentd-based development will come, also to the world of hard real-time systems; but it requires an adaptation of component models and frameworks - a process already on its way for the asynchronous task paradigm, where one may expect to see results in the immediate future. Using the synchronous paradigm in component-based development can in some contexts provide a very attractive solution, but it is unlikely to become very widely spread. Research into systems that consist of (small) synchronous components embedded into asynchronous environment seems like a promising direction. This will form a basis for the development of tools that produce dependable hard real-time systems in the range of small systems, and provide an attractive technology for building components for larger systems. The challenge is to consolidate the results and invent methods and tools that are mature enough to be offered to industry as a future state of the art.

4.6 Level 4 - Quality of Service

Definition

A quality of a system can in general be considered as a function mapping a given system instance with its full behavior onto some scale. The scale may be either qualitative, in particular it may be partially or totally ordered. Or the scale is quantitative, in which case the quality is a measure. The problem of realizing systems that have certain guaranteed qualities, also known as their quality of service (QoS), involves the representation of such qualities in design models or languages and techniques to implement and analyze them as properties of implemented system instances.

While some definitions of 'QoS' include concepts such as security, where the scale is not a measure, we here focus on quantitative measures, especially on those related to time. In this area, there is a common further classification of system requirements, distinguishing between hard real-time requirements, where the quality of any implemented system instance must lie in a certain interval, and soft real-time requirements. Typical examples of such requirements are: "The average lifetime of the battery pack is 4hours", or "The probability of a buffer underrun is less than 0.001". This is the focus of this section, hard real-time systems are handled in the preceding section.

Embedded System Context

Embedded systems designers are usually facing many challenges if they strive for systems with predictable QoS. To incorporate these constraints in the embedded systems design process is a challenging issue, for the following reasons.

- The system dynamics is becoming ever more complex, making it more and more difficult to observe or predict the QoS properly.
- The trend to networked embedded systems raises issues like message buffering, interdependencies due to media sharing, and communication characteristics, all influencing the system QoS.
- Applications involve more and more extra-functional features in the form of multimodal interfaces and multimedia support, having impact on the QoS.
- Modeling and analysis facilities for QoS are (if at all) not well integrated in the methods and tools available to embedded system designer, because QoS relates to different design aspects than the functional design.

For reasons such as these, encapsulation of QoS properties inside a component is very difficult. Work has been done however, mainly on the definition of QoS contracts. A workable approach appears to be to attach 'offered' QoS properties (much like postconditions) to components, as well as 'required' QoS properties (resembling preconditions) [Sel02].

Specifying system and component requirements

Contract Languages. Research has progressed in the context of languages to specify such contracts, and to attach them to component interfaces. We mention QuO/CDL (http://quo.bbn.com), AQuA (http://www.crhc.uiuc.edu/PERFORM/AQuA.html), QML [FK98], and AQML [Nee91]. These are mostly syntactic extensions of CORBA's Interface Definition Language (IDL) tailored to express QoS properties. In order to be useful in component based systems, contract languages must include facilities for expressing properties typical of components, that is, their context dependencies [WBGP01]. A component provides a service under a given contract only if the surrounding environment offers services with adequate contracts. Such dependencies are much more complex than the traditional pre/postcondition contract scheme of object oriented programming.

In the most general case, a component may bind together its provided contracts with its required contracts as an explicit set of equations (meaning that offered QoS is equal to required QoS). Therefore, a component oriented contract language include constructs for:

- expression of QoS spaces (dimensions, units);
- primitives bindings between these spaces and the execution model (bindings to observable events, conversion from discrete event traces to continuous flows, definition of measures);
- constraint languages on the QoS spaces (defining the operations that can be used in the equations, form of these equations).

Verifying Component QoS Properties

In an ideal world, a component user (i.e., a designer that picks a component to include it in a design) has precise information on the QoS behavior of the component with respect to its environment. Then, during component composition, some answers on the QoS behavior of the composed system could be computed.

In practice, contracts written in languages such as QML or QuO are compiled to create stubs that monitor and adapt the QoS parameters when the system is operational. This QoS adaptation software is, in effect, equivalent to a controller for a discrete system. In the approaches practiced today, the following issues limit the confidence that a designer can put in QoS declarations of a component.

- The existing QoS contract languages are not equipped with a formal meaning, thus do not provide a basis for formal proofs, nor can be used to perform symbolic computations.
- The QoS contracts often involve very complex dependencies.
- There are no techniques to prove that a given component implementation abides by the QoS contracts of the component declaration.
- The runtime monitoring cannot fully observe and measure the component's behavior in the defined QoS space, because of technical limitations (e.g., undersampling of events, distributed delay computation).

QoS contract negotiation and adaptation. Components are bound to run in diverse architectures. As a consequence, soft-real time QoS properties are often considered as "promises", and in practice implemented with best effort techniques. QoS contracts are thus not interpreted as final and non-negotiable constraints (differing from the classical interpretation where postcondition failure means bad design). This implies that run-time violations of the contractually agreed QoS can occur. In particular, the component characteristics of "*fully explicit context dependencies*" and the possibility of being able to "*be deployed independently*" are not met by these approaches. Instead the contracts are understood as guidelines for what has to be achieved, and architectural choices by the designer must make provision for variation as well as fallback (minimal) constraints.

Classical component (i.e., non QoS aware) technologies already include facilities for dynamic discovery of resource availability (in other words: level 2 contract negotiation). QoS contract models must support adaptability even further, because a contract may be valid at some instant and invalid at a later time (while level 2 contracts stay valid once "discovered" in a given component execution). Such a support requires means of specifying variation in the QoS contract model, as well as adequate contract monitoring support.

Contract monitoring. Since contracts must be monitored during component execution, the component infrastructure must provide some support to the designer. Building contract monitors is a difficult task, often more difficult than the design and coding of a component implementation. Typical difficulties include

- reliable access to execution events and to precise time for sampling;
- computing with distributed events;
- coordinating distributed monitors, etc.

Therefore, monitors must now be designed by specialists. A component implementation is then augmented by specific pieces of contract monitor. Since time is often an important factor of QoS contracts, the monitor code must be efficiently synchronized with the service code of the component. Aspect oriented programming [KLM+97] and aspect-oriented design [CW02] may provide efficient means of extending a component with contract monitors when those are designed as aspects weaved with the component architecture [HJPN02].

Predicting System Properties from Component Properties

Model-based approaches. We here survey techniques for statically analyzing system performance properties. A workable modeling and analysis approach to embedded systems QoS is based on the observation that networks, interfaces, and even circuits on chips [Con02, Ten00, Ray02] can be understood and modeled as discrete systems exhibiting stochastic behavior, such as error rates, response time distributions, or message queue lengths.

Mathematically speaking, the QoS characteristics of a given embedded system induce families of stochastic decision processes, e.g. Markov chains or semi-Markov decision processes. However, these mathematical objects are too fine grained to be directly specifiable by an average embedded systems designer. Therefore, one must rely on modeling techniques and tools for stochastic processes.

Stochastic modeling and analysis research has given birth to many diverse formalisms, most of them accompanied with tools supporting a QoS-oriented design. This section gives a brief account of the most prominent representatives.

Queuing Networks. Rooted in the early approaches to QoS estimation for analog telecommunication networks, *queueing networks* have since then been used to quantify the quality of many communication system and multiprocessor networks. Queueing networks provide traffic-oriented modeling, where flows of jobs travel through a static structure consisting of queues and processing units [Kle75, Kle76]. Various tools for modeling and analysis of queueing networks exist, such as `Qnap2` (http://www.simulog.fr/eps/mod1.htm), and `Opnet` (http://www.opnet.com/), both being commercial products.

Stochastic Petri Nets. Stochastic Petri nets [Mol82, MCB84, SM91] are extending Petri nets with means to specify stochastic phenomena, and hence allow one to build QoS models. They can alternately be viewed as extension of queueing networks with dedicated means to model resource contention and several other features which are difficult to model in plain queueing networks [Chi98]. In this sense they are more appropriate for contemporary embedded and concurrent system design. Various academic tools exist, among them `GreatSPN` (http://www.di.unito.it/ greatspn/) and `Mbius` (http://www.crhc.uiuc.edu/PERFORM/mobius.html).

Hierarchical models. Modular and hierarchical design has been one of the challenges in QoS modeling. Among the first hierarchical methods is `Hit` [BMW89], which allows one to capture system functionalities and bind it to system resources in a layered approach (http://ls4-www.cs.uni-dortmund.de/HIT/HIT.html). Other methods, including `Quest` (http://www.cs.uni-essen.de/SysMod/QUEST/) [DHMC96] and LQNS (http://www.sce.carleton.ca/rads/ek-rads-etc/software.html) [WHSB98] have developed this idea further. Among others [ESCW01] applies this approach to the UML setting.

Compositional models. Another approach to construct complex QoS models is the compositional one, where systems are incrementally constructed out of smaller components. Typical representatives are `Pepa` [Hil96], `Imc` [Her02], and `Spades` [DKB98]. Tool support exists, e.g. as an add-on to the CADP toolkit (http://fmt.cs.utwente.nl/tools/pdac/) but is not as extensive as for the Petri net-based approaches.

Annotated design methods. Many formal and semiformal design notations have been decorated with QoS characteristics, in order to allow for a QoS prediction on the basis of an integrated model. This approach has been followed e.g. for MSC [Ker01], for SDL [DHMC96], and for Statechart dialects [CHK99, GLM00]. The tools that have appearently been developed for in-house case studies are not publicly available.

Shortcomings. In view of these diverse modeling facilities for QoS, it seems necessary to address the question what shortcomings exists, and what prospect QoS modeling and analysis may have in the coming decade.

- On the principal level, still far too little research is done on the question how to obtain the QoS characteristics to model decisive system aspects or activities. Such characteristics must

be derived from real *measured* system aspects and activities, respectively. For embedded systems, this issue is even more complicated than in other areas, because of the crucial role of the embedding environment.

- On the analysis side, the success of modular modeling methods is variable. While compositional methods so far focus on model construction, not analysis, the layered analysis methods produce notoriously imprecise QoS results (where the inaccuracy relative to the true QoS can be unbounded).
- On a conceptual level, there is still a disturbing gap between the state-of-the-art in functional design, which is component-oriented, and the mainstream QoS analysis tools which are flow-oriented, as in Petri nets or (layered) queueing networks (where tokens or jobs flow through a static structure). This gap – which is also apparent in the relevant UML profile [OMG01b] – hampers a seamless integration in the design process.
- This gap is related to the fact that QoS properties of systems can in general not be deduced from the QoS characteristics of components alone. At least the relevant communication aspects (either direct ones or platform dependent ones) must be taken into account as well.
- The existing QoS contract languages do not possess a precise interpretation that can be used for rigid assessment of contractual obligations.

Research Agenda

The above shortcomings suggest the following research strands to strengthen the development of embedded systems with predictable QoS.

- To enable a modular reasoning about QoS, pre/post condition style contracts should be developed, allowing one to specify interfaces with *'required' vs. 'guaranteed' QoS* [Sel02]. Since virtually any QoS measure is a stochastic quantity, both QoS guarantees and QoS requirements must be expressible in a probabilistic setting (a simple example would be: 'in 95answer must come within 3 seconds')– as opposed to an absolute setting ('the answer must come within 3 seconds'). In full generality one needs means to express those quantities via probability distributions which may be parameterized by the component input.
- Due to the fact that QoS characteristics of a system cannot be derived only from QoS characteristics of components leads to the necessity to reconsider the architectural approaches in such a way as to at least reduce the dependencies of different parts of systems.
- The size and complexity considerations sketched in the beginning of this section ask for major research endeavors with respect to management of the design for predictable QoS. Efforts must be undertaken to strengthen a compositional reasoning with probabilistic quantities. The layered analysis approach is going in this direction, but is too inaccurate. Better, and more manageable, compositional methods are needed.
- A seamless QoS design process relies on a smooth but solid integration of the QoS modeling and analysis concepts into a well-designed integrated formalism, which is semantically deeper than a shallow annotational extension of the UML. Put differently, we need QoS contract languages with precise semantics. A seamless design process also requires that QoS design and engineering principles must be transferred from a flow-oriented to a component-oriented modeling paradigm.
- Industrial-strength tools supporting this integrated QoS design process are needed. These tools are expected to emerge as extensions of existing component-oriented modeling and analysis tools.

4.7 Reliability

Reliability is discussed in Section 4.1 of the Action 1 HRT Roadmap. We will therefore here be content with a short account.

Definition. A technical definition of reliability can be given in terms of the probability that the system work as intended (provide its service) during a certain time period. Two major techniques for reliability analysis are fault trees and stochastic analysis using Markov chain. Stochastic analysis can be carried out using techniques that have been discused in the previous Section 4.6. Here, we will just indicate how they can be specialized to consider reliability analysis.

Description of Component Properties. A standard approach is to associate components with a reliability model, which involves fault events, error states, fault arrival rates, and a Markov chain model of how the component responds to fault events.

Analysis of System Reliability Properties. The system description must describe how errors propagate between components. Then the reliability models of individual components can be combined into a global Markov chain, which can be analyzed using a separate tool for Markov chain analysis. This approach is used in the MetaH toolset, where the SURE/PAVE/PAWS tool (from NASA LAngley) is used to solve the resulting Markov chain.

4.8 Specifying and Reasoning about Contracts: Summary and Analysis

From the various information of this Section 4, it is clear that the main difficulties of the contract-based specification, verification and validation fall into a few general categories:

- **Harmonization of Specfication Techniques:** Current contract-based specification techniques use notations and models that are quite different. In order to fully support all aspects of component based design, these notations and models must be harmonized. This not a simple task: for instance, crossbreeding notations and research results on behavior specification and performance analysis is not obvious; one needs a time model that is compatible with the behavior notation as well as the component framework.
- **Obtaining Extra-Functional Properties of Components:** Timing and performance properties are usually obtained from components by measurement, usually by means of simulation. Problems with this approach are that the results depend crucially on the environment (model) used for the measurements may not be valid in other environments, and that the results may depend on factors which cannot easily be controlled. Techniques should be developed for overcoming these problems, thereby obtaining more reliable specifications of component properties.
- **Obtaining Component Properties by Static Analysis:** Functional, and some non-functional component properties can in principle be inferred by static analysis of the software itself, although it is often difficult in practice. Some interesting recent advances concerning synchronization and timing properties prompt for further work to enlarge the scope of static analysis techniques.
- **Dependency on the Underlying Platform:** Many component properties are highly dependent on its environment, including other parts of the system as well as the system platform. It is highly non-trivial to express component properties in such a way that these properties can be applied in a variety of environments. As an example, properties of execution times of components depend crucially on the timing properties of the underlying platform. There are currently no widely usable solutions for specifying the timing behavior of a component in a platform-independent way.

- **Techniques for Building Run-Time Monitors:** Contracts must in general be monitored during component execution, unless there are guarantees that a component satisfies a contract. For embedded systems, techniques for (automatically) constructing *efficient* (using limited resources) monitors from contracts must be further developed. Aspect oriented programming [KLM+97] may provide structuring techniques for adding monitors to a system.
- **Diagnosing Causes of Poor System Properties:** When the predicted system properties do not meet requirements, it is important that analysis techniques can point at the "bottlenecks" in a design, i.e., the components that "cause" poor system behavior.
- **Guidelines for Tractable Analysis of System Properties:** The verification and prediction of system properties from component properties is in the general case an intractable problem, i.e., general techniques can cope only with systems of small or medium complexity. To master this complexity, we need guidelines for structuring assemblies, in other words software methodologies that help the designer to build "tractable" architectures by enforcing well chosen restrictions.
- **Adapting Well-Understood Design Principles:** Many advances in component technology have been obtained by adaptation of well-understood design techniques to the component-based setting. An example is the use of classical schedulability analysis and reliability analysis in some component technologies for real-time systems. There are many other practical techniques that could potentially be adapted to and enrich component technology.
- **Relating Generic Contracts to Specific Platforms:** Existing component models and their supporting frameworks may often rely on particularities of the underlying platform. Hence they do not necessarily match the platform independent, contract-based specification techniques. There is therefore a need to bind platform independent notation, techniques and tools to platform dependant models. This is a difficult issue, because component infrastructures are very different fom one application domain (eg automotive systems) to another (eg network based information systems). Each class of platform brings specific difficulties; for instance a network model may not match a real protocol implementation on some platform, from both the behavioral and quality of service points of view. One potential solution to this platform dependency problem could be the implementation of architecture transformations along the lines of the MDA approach.
- **Tool Support:** must be developed to support the tasks described in this section.

5 Component Models and Integration Platforms: Landscape

This section is an overview of existing component models and component integration platforms. The emphasis is to describe the component models themselves, and their support for handling the aspects that are described in Section 4. In some cases, particularly for proprietary component models, there is a tight connection between the component model and a particular integration platform, and sometimes also to a particular ADL. We then describe separately the model itself. Thereafter follow descriptions of (some) available toolsets and platforms that support this component model.

The emphasis of the overview is to shortly describe for each model, how it supports the aspects that were described in the previous section.

- Component Model, including its support to describe different properties in component interfaces
- Composition
- Verification and validation tools and techniques
- Supporting integration platforms

5.1 Component Models for Embedded System Design.

First in this overview, we survey component models and platforms that have been developed for application to embedded systems. Typically, component implementations are given in a compilable language, (C being the most common one) and are composed before compilation. Execution semantics are given by a run-time executive or a simple RTOS.

5.1.1 Programmable Logic Controllers: the IEC 61131-3 standard

Introduction
In the area of Industrial Automation, PLC's (Programmable Logic Controllers) are a widely used technology. However, for the last twenty years, the corresponding applications have been written in many different languages, resulting in inefficient work for technicians, maintenance personnel and system designers. For instance, there are numerous versions of the so-called ladder diagram language, and furthermore this language is poorly equipped with facilities such as

- control over program execution,
- definition and manipulation of data structures,
- arithmetic operations or
- hierarchical program decomposition.

These problems led to the constitution of a working group within the IEC (International Electrotechnical Commission), with the aim to define a standard for the complete design of programmable logic controllers. While previous efforts had been made before, IEC 61131 has received worldwide international and industrial acceptance. The first document introducing the general concepts was published in 1992 and followed by the definition of equipment requirements and tests. The core of the standard is its third part, published in 1993, which describes the harmonization and coordination of the already existing programming languages. The 8 parts of the standard can be purchased at http://www.iec.ch.

Note that, due to the various types of hardware, the aim is not toward a single programming system for all controllers. Instead, certified IEC 61131-3 programming systems have an agreed degree of source code compatibility and have a similar look and feel. Yet they will differ in debugging features, speed, etc.

An international organization of users and producers, PLCopen (http://www.plcopen.org), was founded in 1992 with the aim of promoting the development and use of compatible software for PLC's. PLCopen offers tests for IEC 61131-3 compliance, but also a course, designed for experienced or beginner PLC programmers who want to develop software according to IEC 61131-3 and for support and implementation engineers who modify systems programmed according to IEC 61131-3. There are also smaller users-only organization, e.g. EXERA, which propose tests for the compliance of programming environments.

Short Technical Description.

- **Component types:** An application is divided into a number of *blocks*.
- **Supported languages:** A block is written in any of the languages proposed in the standard. There are two textual languages (ST, IL) and three graphical languages (FBD, LD, SFC).
  - 1) **Function Block Diagram (FBD)** is used for the description and regulation of signal and data flows through function blocks. It can nicely express the interconnection of control system algorithmics and logic.
  - 2) **Structured Text (ST)** is a high level textual language, with a Pascal-like syntax.
  - 3) **Instruction List (IL)** is an assembler-like language, found in a wide range of PLC's.

  - 4) **Ladder Diagram (LD)** is a graphical language based on the relay ladder logic, which allows the connection of previously defined blocks. For historical reasons, it is the most frequently used in actual PLC programs.
  - 5) **Sequential Function Chart (SFC)** is used to combine in a structured way units defined with the four languages above. It mainly describes the sequential behavior of a control system and defines control sequences that are time- and event-driven. It can express both high-level and low-level parts of a program.

- **Visibility of Underlying Hardware:** While it aims at enhancing portability of PLC programs, the IEC 61131-3 has several features referring to the actual underlying hardware (variables can be linked to physical addresses, etc.)
- **Syntactic Support:** Each functional block has a set of in-ports and out-ports. IEC 61131-3 also requires **strong data typing** and provides support to **define data structures**, which can be used to transmit information as a whole between different units. More precisely, while a function simply computes its output from its input, without internal variables, a function block consists of a set of data, together with the algorithms handling these data, a bit like the definition of class in an object-oriented framework (no further comparison can be made, though). Input and output parameters must be formally defined, to ensure a clean interface between different function blocks. This notion thus appears as an important feature, meant to encourage the development of well-structured software: blocks can be viewed as the basic components of a program. Since it is re-usable within a given program, but also from outside, an increased use of such blocks will lead to the construction of powerful libraries.
- **Support for Functional/Extra-Functional Properties:** Function block execution may be periodic or event-driven. There is no support for analyzing other properties than syntactic properties.

Tools for the IEC 61131-3 Standard
Today, all large suppliers of PLC's have announced IEC 61131-3 compliant development systems. They propose different programming environments (prices up to 3 Keuros) for code generation toward given hardware, with some architectural aspects as parameters. For instance:

- Siemens with STEP7
- Allen Bradley with Control Logic
- Schneider-Electric with PL7PRO.

There are also smaller suppliers, either for PLC's only, or for programming platforms only. Of course, due to developments in the industrial market producers may not always be 100% standard compliant.

Assessment and Further Needs

The IEC 61131-3 standard is widely spread. Compared with traditional programming systems, it appears to be a major step forward. The new set of languages is said [Lew98] to significantly improve the quality of PLC software, and in particular to overcome the weaknesses of previous versions, especially about the above mentioned Ladder Programming. The improvement also concerns the communication and software model. Finally, a major benefit for end-users using IEC 61131-3 compliant products will be inter-system portability.

However, the IEC 61131-3 standard is not fully mature and the portability issue remains an important problem. For instance, users feel the need to have textual files, that can be used to connect different platforms. Furthermore, some ambiguous semantics remain for the languages. Finally, new requirements emerge: systems will become more distributed with more parallel processing. Therefore, new standards are under development, such as the function block standard IEC 1499, not to replace the former but to work in conjunction.

5.1.2 Koala

Introduction
Koala is a component model and an architectural description language that successfully works for consumer electronics devices. Koala is developed and used at Philips. It was designed to build software control units for consumer products such as televisions, video recorders, CD and DVD players and recorders, and combinations of these (e.g. a TV-VCR). Koala is currently in use by a few hundred software engineers for the creation of a family of televisions. More information of Koala can be found in [vO02], [vOvdLK00], [FEHC02].

Short Technical Description.

- **Component types:** A Koala component is a piece of code that can interact with its environment through explicit interfaces only. As a consequence, a basic Koala component has no dependencies to other Koala components.
  - **Component Implementation** is a directory with a set of C and header files that may use each other in arbitrary ways, but communication with other components is routed only through header files generated by the Koala compiler, based upon the binding between components.
  - **Component Interface:** the directory also contains a component definition file, describing among other things the interfaces of the component.
- **Visibility of Underlying Hardware:** The Koala component model itself is done on an abstraction level that is independent of hardware. The hardware dependency is encapsulated in particular components. The entire development environment is adjusted for development of particular product families which improves the efficiency of the development process, but looses on the generality.
- **Syntactic Support:** Connections between components are expressed in terms of interfaces which are implemented as a small set of semantically related functions. Koala model identifies two types of interface: *provides* interface and *requires* interface. Koala provides interfaces are similar to the interfaces as known from COM and Java. A component may provide multiple interfaces, which is a useful way of handling evolution and diversity. Koala requires interfaces identifies interfaces of other components and interface required from the environment of the component. All communication is routed through such requires interfaces. Koala interfaces can be optional. An optional requires interface need not be connected - an optional provides interface need not be implemented. This allows components to fine tune themselves to their environment, by observing what the environment can and cannot deliver.

  Connectors connect requires interfaces of one component to provides interfaces of another component. Naturally, in compound components it is also possible to connect provides interfaces of subcomponents to provides interfaces of the compound component, and similarly for requires interfaces.
  - **Interface Compatibility** It is allowed to connect a requires interface to a provides interface of a wider type. The provides interface should implement all of the functions of the required interface, but it may implement more than that.
  - **Glue Code** can be added to the binding between interfaces. Simple glue code can be expressed in an expression language within Koala; more complicated code can be written in C. This allows to easily to overcome a certain category of syntactic and semantic differences. A special case of glue code is code that switches a binding between components. Such a mechanism to select between components can be implemented in C, but it occurs so frequently that a special concept for this is defined in the language: the switch. The compiler converts a switch internally to a set of Koala expressions, which has the advantage that it can perform certain optimizations, such as reducing the switch to a straight binding if the switch is set to a position that is known at compile time. The binding through the glue module and the switch are examples of connectors. The Koala language defines no other connectors.

- **Support for Behavioral Properties:** No extra-functional properties are specified in the interface of components. In a system design, it is possible to specify the ordering of tasks using precedence relations and mutual exclusion. There is also support for deriving some of the system properties form the components. For example memory consumption of the system is calculated form the memory consumptions of the components, which is a parametrised value.
- **Support for Timing Properties**. No support for timing properties.
- **Support for Performance Properties**. There is no support for modeling performance properties.

Short information about Koala tools

- **Supported languages for component implementations:** A component is an implementation in C language by using specific rules. Koala uses a "Koala language" for constructing applications from the components by connecting the interfaces of the components.
- **Supported development platforms:** A proprietary development platform that includes Koala language and compiler which compose components (C code) exists.
- **Supported target platforms:** Proprietary platforms.
- **Status:** The use of Koala is growing within Philips. There are plans for building additional development environment tools, such as visual composition and visula component selection.
- **Availability:** There are plans to publish Koala as open source.
- **Degree of Automation:** The Koala compiler collects the involved components and makes some optimizations such as removing unused interfaces and resolving connections of conditional types. The component binding is of static type solved by generation of C code. The compiler can also optimize the memory usage of the application (the so-called footprint), by eliminating functionality in a component that is not used. Most of the documentation (header files, etc.) must be created manually. This has not been seen as a large overhead, although there are plans to improve this process by building a set of supporting tools.
- **Run-Time Infrastructure:** As in many small embedded systems, a system using Koala component model is a single-process image, built on a top of a small real-time kernel with pre-emptive scheduling, which separates the high frequency tasks from the low frequency tasks. Separate activities can be allocated to light-weight threads, which are managed by the kernel.
- **Analysis Support for:**
  - **Memory Footprint:** The Koala component model and its implementation to some extent allow calculating and predicting resource consumption. For example, **memory consumption** can be estimated at composition time (compile time), and this feature is built in the Koala compiler. For technical details see [FEHC02]. Using this calculation model it is possible to budget the memory for particular components and by a parameterization of the interface define the particular properties of the components.
- **Library Support:** Koala components are stored in a repository. The repository is a set of packages, where each package is a set of component and interface definitions, some of which are public, some of which are private.

Summary
The Koala component model is an example of implementation of a component-based approach that works successfully in a large industry. It is a good example of an evolutionary approach in building support for component-based development. The design and implementation fulfills the following requirements [CL02, Ch. 12] related to the component-based approach.

- It devises a technique with which components can be "freely" composed into products, as the main approach to deal with diversity in a product population. The technique must work in resource-constrained environments such as televisions and video recorders (which are typically 10 years behind PCs in computing power).
- Make the product architectures as explicit as possible, to manage complexity.
- Let components make as little assumption as possible about their environment.
- Allow for parameterized components that are "when instantiated" as efficient as dedicated components.
- Allow for various ways of connecting components; more specifically, allow for adding glue code to the connection between components.

These requirements are valid for many embedded and RT systems. Does it mean that the Koala component model is so general that it is possible to use it in other domains? In many aspects this is true. The basic principles, which are derived from the basic principles of CBD, are valid in general for embedded systems. In implementation, in some parts the domain knowledge is implicitly built in (due to the various reasons, one of them to improve the development efficiency and the performance). In order to use the Koala model as a general component-model for embedded systems, some parts should be removed or explicitly separated as domain-specific. The Koala component model provides good bases for further improvement of achieving predictability of extra-functional properties.

The strong points of Koala are:

- Separation of the provided from the required interfaces of a component.
- Interaction with the environment, including the underlying hardware-dependent services are obtained exclusively via the interfaces.
- There is a strict definition of the component development process, including quality assurance, and a form of component certification.

The weak points of Koala do not lie in the component model itself, but to a large extent in the lack of tools supporting efficient development in large scale. Currently, Koala developers must conform to rules that can be violated, unless checked automatically. Potential tool functionality could include the following.

- tools that manage components (component repository, component browsers, visual environment, etc.)
- checks that a component has no other dependencies than through its explicit interfaces
- generation of glue could to some extent be automated.
- Support for analysis and composition of timing and performance (and some other properties) is rudimentary and can further be developed. An obstacle is that many of these properties are hardware and platform dependent and cannot be a part of a general model.

5.1.3 Rubus Component Model

Rubus is a small Real Time Operating System, developed by *Arcticus Systems AB* (http://www.arcticus.se/). Rubus is divided into one part supporting time-triggered execution and one part supporting event-triggered execution. Time-triggered execution is to support hard real-time applications with a deterministic execution mechanism. In order to supports component-based development of hard real-time systems has Arcticus Systems AB together with Department of Computer Engineering

at Mlardalen University developed a component model and associated development tools for use with the Rubus operating system [IN02]. The component model is used in projects within Volvo Construction Equipments Components AB. We include a description of this model, in order to illustrate how a component model can be developed on top of an execution platform in order to suit particular needs.

Short Technical Description.

- **Component types:** A basic Software Component consists of a behavior, a persistent state, a set of in-ports and a set of out-ports and an entry function. Its main functionality is given by an *entry function*. A *task* provides the thread of execution for a component. The entry function takes as an argument a set of in-ports, the persistent state, and the reference to the out-ports. In [NGS+01], it is stated that entry function code may not contain any call to communication services. Instead the compiler that compiles the system description automatically generates the communication infrastructure. For example, assume that an out-port of a component A is connected to an in-port of a component B, the generated code (system task) will assure to copy the information automatically under the given synchronization and timing requirements. The **attributes** of a task are *Task Id*, *Period*, *Release Time*, *Deadline*, and *WCET*. In addition, precedence and mutual exclusion ordering between tasks can be specified.
- **Visibility of Underlying Hardware:** System descriptions do not mention hardware configurations, they are intended to be used by a schedule synthesis tool.
- **Syntactic Support:** Each Component has Input Ports and Output Ports for communication. Tasks in the safety-critical part communicate without buffering. There is a type system for data
- **Support for Behavioral Properties:** No functional properties are specified in the interface of components. In a system design, it is possible to specify the ordering of tasks using precedence relations and mutual exclusion.
- **Support for Timing Properties**. The timing requirements are specified by release-time, deadline, WCET and period. There is a tool for schedulability analysis.

Short information about Rubus tools

- **Supported languages:** The functionality of a system can be mode-dependent. Temporal coordination between tasks is specified for each Mode by a *software circuit* or *dataflow model* which specifies the output-input connections between tasks, and timing constraints on tasks and their composition. Precedence/exclusion information can also be included.
- **Supported languages for component implementations:** C
- **Supported development platforms:** Available on Windows and Linux platforms
- **Supported target platforms:** Rubus OS is ported to a number of target and program development tools.
- **Status:** Commercial product.
- **Degree of Automation:** A tool designated "Rubus Visual Studio" exists, which manages the components available and their associated source files, so that components can be fetched from a library and instantiated into new designs.
- **Analysis Support for:**
  - **Timing Properties:** Scheduling is derived automatically from the ADL description, using task attributes and precedence/exclusion information. Time-triggered tasks are statically scheduled, and event-triggered tasks are scheduled on-line by fixed-priority pre-emptive scheduling.

  - There is currently **no support** for **performance properties**, **reliability** or **safety analysis**.

- **Support for Distribution:** Rubus supports distribution, over buses that supports time synchronization (such as TTP and TTCAN)

Summary

Rubus is an example of how a component model can be developed on top of an existing RTOS. The model makes system integration easier, by allowing timing analysis to be performed based on a system description. The development of the Rubus component model has been a significant improvement for software development inside Volvo CE. For future development, the Rubus platform may face a shortage of tools that support component-based system development, in a similar way as was discussed for the Koala model. There are currently no automated checks for

- checking that components use only the explicit interfaces for communication,
- WCET (worst-case execution time) analysis of components, nor for
- allocation of tasks to processing nodes.

5.1.4 PECOS

The PECOS project (http://www.pecos-project.org/) [CL02, NAD+02, WZS02, PEC], funded by the EC under the IST Program (project number: IST-1999-20398), aims to enable component-based software development for embedded systems such as smart cell phones, PDAs, and industrial field devices. In order to validate component-based software development (CBSD) for embedded devices the project has developed the hardware and software for a field device as a case study for embedded systems with real-time requirements.

The project has included four main activities:

- **CBSD processes** The PECOS process aims to enable CBSD for embedded systems, specifically for field devices. It addresses the major technological deficiencies of state-of-the art component technology with respect to extra-functional requirements, such as limited CPU power, memory and hard real-time.

- **Component Model:**
  - **Interfaces** are defined by *Input Ports* and *Output Ports*, and *connectors* connect compatible ports. Ports have basic types.
  - **Component Types:** Active Components (with own thread), Passive Components (encapsulating behavior without threads), Event Components (triggered by events)
  - The **attributes** of a Component can specify Memory Consumption and WCET, cycle time, priority
- **ADL:** The **CoCo Component Language** is used for the specification of components, entire embedded devices, and architecture and system families. In CoCo, a composite active component (with thread) specifies the execution rules (called *schedule*) for its subcomponents.
- **Lightweight composition techniques** The CoCo Component Language is used for the specification of components, the specification of entire embedded devices and the specification of architecture and system families. CoCo provides the concept of abstract components, composition rules to allow composition checking.

- **Platforms and tools** A translation from CoCo to target languages such as C++ and JAVA has been developed. The PECOS model is mapped to a prioritized, pre-emptive multi-threaded system to realize the different components, passive, active and event. A technique has been introduced to enable data exchange between components. The developed tools are embedded in the open source framework ECLIPSE as plug-ins.

Other characteristics:

- **Supported Languages:** Includes a composition language CoCo that is translated to C++ and Java. A CoCo component structure is mapped to an identical class structure. Connectors are mapped to shared instance variables in the enclosing object. Ports map to set and get methods.
- **Scheduling:** The model does not specify anything regarding the scheduling of components, what scheduler can be used and how schedules can be checked to see if they are actually feasible. It only assumes that there is a scheduler.
- **Availability:** embedded in the open source framework ECLIPSE

  http://www.eclipse.org/ as plug-ins. It can include any proprietary integration platforms (developed by companies such as ABB, Boeing, Dassault, EADS, Thales). So far ABB has started to integrated the model with its proprietary platform.

Summary

The PECOS is unique in the sense that it addresses several aspects of component-based software engineering - development and life cycle process, component model that treats temporal and other extra-functional attributes, architectural modeling and development tools. However the model is still not fully developed and it remains to be seen how successful its implementation will be.

5.2 Component Models for Run-Time Composition

In this subsection, we survey component models and platforms that have been developed not primarily for embedded systems. The deployment and composition of components is typically performed at run-time.

5.2.1 Java Beans

Sun Microsystems initially introduced a client-side (or desktop) component model (JavaBeans), and subsequently a server-side (or enterprise) component model (Enterprise JavaBeans). Both of these build on a Java-based approach to distributed applications [Mic02]. In the JavaBeans specification a *bean* is a reusable software component that can be manipulated visually in a builder tool; this differentiates beans from class libraries which cannot benefit from visual manipulation even if providing the equivalent functionality. Some of the unifying features of JavaBeans are the support for property customisation (to control the appearance and programmatic behavior of beans), event handling (a communication metaphor based on delegation and event listeners), persistence (serialisation of a bean's state for later reloading or transmission over a network) and introspection (analysis and manipulation of a beans internal structure, e.g. properties, events, methods and exceptions). A major quality of the Java Beans component model is its simplicity - the specification of the model is only a 114-page document [Mic97].

JavaBean was designed for the construction of graphical user interfaces (GUIs). Customization of components plays an essential role in JavaBean and was originally emphasized to enable incremental specialization of GUIs from generic exemplars.

Short Technical Description.

- **Component types:** A *bean* is defined as a self-contained, reusable software unit that can be visually composed into applets, applications, servlets, and composite components, using visual application builder tools.

  Programming a JAVA component requires definition of three sets of data: (i) properties (similar to the attributes of a class); (ii) methods; (iii) events that are an alternative to method invocations for sending data. A bean wishing to receive an event (listener) registers at the event source (component launching the event). In Java, events are objects created by a source and propagated to all registered listeners. Example: An alarming device (listener) asks a temperature probe (source) to send it a message when temperature value exceeds a certain threshold so that a bell can be sounded.

- **Syntactic Support in Interfaces:** The model defines four types of interaction points, referred to as *ports*:

  – methods, like a method in Java.

  – properties, used to parameterize the component at composition time or as a regular attribute in the OO sense of the term during run time.

  – event sources, and event sinks (called listeners) for event-based communication.

- **Other Support in Interfaces:** JavaBeans does not support behavioral or QoS properties.

- **Support for Introspection:** A bean can be dynamically queried for its characteristics (operations, attributes, etc.), through an introspection mechanism available to all Java classes. The component programmer can, however, restrict the amount of information made available. To do so, the Java component implements a *BeanInfo* interface whose methods return a list of properties, operations, etc.

- **Supported Programming languages:** code must be written in Java.

- **Required "Middleware"/Framework Support:** None.

Short information about JavaBean tools

There are several commercial programming environments, e.g., Sun's Net Benas, IBM"s Visual Age, and Borland's JBuilder. These builders build assemblies visually as a graph of components, where ports between beans are connected. Note that the JavaBean component model by itself does not specify how to connect components; this is done by the builder tool.

5.2.2 EJB - Enterprise Java Beans

The EJB model is a server-side component model, which is rather different from JavaBeans. The EJB specification defines its component architecture in terms of a scalable environment, based on *containers* (see below), that provides runtime services for managing component activation, concurrency, security, persistency and transactions [JF00]. The EJB specification defines a component model by standardising the contracts (context and callback interfaces) and services offered by the runtime environment, and the patterns of interaction between components.

Short Technical Description.

- **Component types:** There are three types of EJB beans defined:

  – **Entity beans** are application elements that embody data and are by nature transactional (and persistent); these beans may handle the persistency themselves (bean-managed persistence) or delegate it to the container (container-managed persistence).

  – **Session beans** are used to model business processes in a transactional and secure manner without the need for persistent storage (i.e. they last for the duration of a session).

  - **Message-driven beans** are created by containers to asynchronously handle messages from the Java Messaging Service (JMS) sent, for example, to a queue transparently associated with the bean.

- **Syntactic Support in Interfaces:** Depending on the bean type, developers must implement associated, pre-defined, callback interfaces (e.g. EntityBean, SessionBean, MessageDrivenBean). These callback interfaces are used by containers to manage and notify beans about certain events (e.g. bean activation or passivation, instance removal, transaction completion, etc.). Moreover, each type of bean expects a specific interface or context from the container (e.g. EntityContext, SessionContext, MessageDrivenContext) for getting an entity beans primary key, identifying the bean caller, transaction demarcation, etc.

- **Other Support in Interfaces:** The container provides a uniform interface with services such as naming (Java Naming and Directory Service), security (public/private key authentication and encryption), transactions (based on Java Transaction Service or OMG Object Transaction Service), and messaging (Java Messaging Service).

- **Support for Introspection:** A bean can be dynamically queried for its characteristics (operations, attributes, etc.), through an introspection mechanism available to all Java classes. The component programmer can, however, restrict the amount of information made available. To do so, the Java component implements a *BeanInfo* interface whose methods return a list of properties, operations, etc.

- **Supported Programming languages:** code must be written in Java.

- **Required "Middleware"/Framework Support:** Containers act as a level of indirection between clients and bean instances. Each container provides objects, called EJB objects, that expose bean functionality and intercept every method call before delegating it to the bean. This EJB object is automatically generated and embodies container-specific knowledge about bean activation, transactions, security and networking [RAJ01]. Additionally, each EJB object implements the remote interface that enumerates all business methods exposed/implemented by the respective bean.

  EJB relies heavily on Java Remote Method Invocation (RMI) platform to support dynamic class loading, automatic activation, remote exceptions and distributed garbage collection [RAJ01]. RMI is a distributed architecture which uses a RPC-based protocol supporting inter-process communication. For example, EJBHome and EJBObject remote interfaces rely on the RMI infrastructure for transparent distribution of component functionality.

- **Deployment:** The deployment of beans is in terms of EJB-jar files that package bean classes, remote interfaces, home interfaces and bean properties files, together with an XML-based deployment descriptor that contains information on all the packaged beans (e.g. home name, bean class name, primary key, container fields, etc.) and their dependencies. Deployment descriptors also declare the middleware services needed by components (e.g. life-cycle policy, persistency handling, transaction control), thus avoiding the non-standard manifest files that were previously used with JavaBeans. [RAJ01].

Short information about EJB tools
The Java Development Kit (JDK) includes a set of classes and development tools (e.g. an RMI compiler) to support the automatic generation of distribution classes (e.g. stubs), glue code (e.g. container policies) and deployment descriptors, thus alleviating developers' efforts and responsibilities.

Summary EJB standardises a distributed architecture for building component-based business applications. EJB builds on the previous JavaBeans client-side component model and particularly on the Java language, and essentially realises the WORA principle (Write Once Run Anywhere) (albeit in a language dependent manner). EJB also relies on the RMI architecture, and simplifies and automates the development and distribution process. In contrast to CORBA (see below), RMI not an open standard (although it has the advantage over CORBA that it fully supports

objects passed by value (via serialisation). However, efforts have been made to make EJB portable to CORBA systems, particularly through standard connectors (vendor specific bridges that link different architectures) along with RMI-IIOP mappings [RAJ01]. Most crucially, the EJB software engineering process is grounded on a set of tools and code generators that automate the development and deployment process by hiding the cumbersome details of handling distribution and component management policies (e.g. life-cycle, security, transactions, persistency).

5.2.3 COM, DCOM, COM+

COM (Component Object Model) [Cor95] dates back to 1995, and is typical of early attemps to increase program independence and allow programming language heterogeneity. COM has roots in Microsofts OLE (Object Linking and Embedding, first version in 1991), which provided a standard way to embed or link data objects (e.g. text, graphics, images, sound, video, etc.) inside document files, hence supporting the creation and management of compound documents. Unlike EJB and CORBA (see below), COM provides a binary solution to interoperability and extensibility [Bro96, Gos95] (see below). The Distributed Component Object Model (DCOM), which supported inter-process communication across distributed machines through an RPC-based protocol called Object RPC (ORPC) [Pat00], was introduced with Windows NT 4.0. More recently, the component model was extended (and renamed COM+) and integrated with Windows 2000 to support the development, configuration and administration of distributed systems with automatic and integrated (Windows-based) control over several aspects of business applications (e.g. security, synchronisation, transactions, queues and events).

The COM component model is fundamentally an intra-address space model. COM extends object-oriented design principles by hiding a components implementation behind its interface(s) (encapsulation) and allowing components to be replaced by different implementations of the same set (or super set) of interfaces (polymorphism) without the need to recompile their clients. These principles are possible in COM because component services (collections of interfaces) are separated from their implementation through a binary-level indirection mechanism called a virtual table (cf. C++ virtual functions or vtables) [Szy98]. This is the basis of the binary solution referred to above. At runtime, an interface is a typed pointer (known as an Interface Identifier - IID) to a specific virtual table that references the functions (methods) implementing the services exposed by the interface. This binary interface structure allows interoperability between software components written in different languages as long as compilers can translate language structures into this binary form [Gos95].

Short Technical Description.

- **Component types:** A COM component can be seen as an object at the binary level, whose implementation is clearly hidden behind its interface.
- **Syntactic Support in Interfaces:** COM defines interfaces on the level of binaries, consisting of data and function declarations. There are standard protocols for calling an interface, and for dynamically discovering and creating objects and interfaces.

  Independent development brings the potential for naming conflicts between interfaces and their implementations. To avoid this, COM requires developers to assign a unique Interface Identifier (IID) and Class Identifier (CLSID) to each newly specified interface and class implementation, respectively.

  COM does not support inheritance; basic component composition is available through

  - **containment**, where a COM object contains other COM objects: the outer object declares some of the inner objects interfaces; at run-time it delegates calls to these interfaces to the inner objects,
  - **delegation**, this employs wrappers that insert behavior before or after delegating the method calls to inner classes,

  - **aggregation**, where the interface of the inner object is exposed without the overhead of call indirection; aggregation requires the source code of both the inner and outer objects to be changed.

- **Other Support in Interfaces:** COM does not support behavioral or QoS properties.
- **Support for Introspection:** Components can be queried to find out their supported interfaces.
- **Supported Programming languages:** code can be written in any programming language as long as the compiler generates code that follows the binary interoperability convention. Component interfaces are defined using the Microsoft Interface Definition Language (MIDL) which is an OSF/DCE-based adaptation of CORBA IDL. The MIDL compiler generates marshalling classes and type information (e.g. proxies, stubs, header files, type libraries) needed to accomplish binary compatibility - i.e. joint deployment of components developed in different languages.
- **Required "Middleware"/Framework Support:** Component interfaces are separated from their implementation through an indirection mechanism called a virtual table (cf. C++ virtual functions or vtables) [Szy98]. At runtime, an interface is a typed pointer to a specific virtual table that references the functions (methods) implementing the services exposed by the interface. The framework needs a run time engine that creates COM objects. COM objects are also automatically garbage collected.

  DCOM (1996) extends COM with distribution, based on the DCE RPC mechanism. The component model itself is unchanged.

  COM+ (1999) is an extension of DCOM with the container approach (see text on EJB and CCM in sections 5.2.2 and 5.2.5), using the runtime platform MTS (Microsoft Transaction Server). The container intercepts calls to a component, and can execute pre- and post-processing actions to implement various services. Typical services offered include support for transactions, concurrency control, load balancing, role-based security checks.

Short information about COM/COM+ tools

The COM framework is rather specific to Windows platforms (although it is also implemented on VxWorks). It is supported by several tools for development on Windows platforms, such as Visual Studio.

Summary

The COM+ component model focuses on the provision of enterprise distributed applications. It tries to take domain-neutral aspects out of source code and expose them through declarative attributes that can be used to control service context (e.g. process synchronisation, security profiles, automatic transactions) [Box00]. Nevertheless, it is not possible to add new attributes, hence making this mechanism limited for the majority of COM developers. Additionally, COM+ type information management is rather cumbersome, i.e. it uses disparate information formats (e.g. IDL, type libraries, MIDL-generated strings embedded in proxy DLLs), sometimes with no mappings between them. Furthermore, COM+ runtime type information (type library) permits us to advertise only the types exported by a component but not component dependencies [Box00].

5.2.4 .NET

The .NET component model and framework allows to write applications in different proramming languages. .NET also provides a run-time platform with a number of services. .NET does not rely on COM, because binary interoperability is too limited. Instead, a .NET a compiler translates source code into an intermediate language called the Microsoft Intermediate Language (MSIL), very similar to the Java Byte Code. The common language runtime (CLR), very similar to a Java

Virtual Machine then takes the intermediate language and, on the fly, converts it into machine-specific instructions.

The CLR is able to recognise and execute portable executable (PE) files, which are image files that combine MSIL code with metadata (stored in metadata tables and heaps). This approach avoids the need for multiple and disparate metadata formats (e.g. type libraries, header and IDL files) and enables the generic use of reflection, serialisation and dynamic code generation in a type safe manner [MG02].

MSIL compilers are responsible for automatically emitting metadata into the PE file (e.g. information describing types, members, references, inheritance, etc.). The runtime environment then uses this binary metadata information (cf. managed data) to locate and load classes, control memory usage, resolve invocations, manage runtime context boundaries, enforce security and compile to a particular computer architecture by using specific just-in-time (JIT) compilers. Metadata is the .NET language neutral way to provide binary information describing: assemblies (e.g. unique identification, dependencies on other assemblies, security permissions), types (e.g. base classes, implemented interfaces, visibility), members (e.g. methods, fields, properties, events) and attributes (extra metadata modifying the properties of types and members).

.NET addresses programming of services for Web-based software development. For this purpose, The .NET framework is complemented by a set of unified class libraries for standard programming (e.g. I/O, math, etc.), for accessing operating system services (e.g. network, thread, cryptography, etc.), for debugging and for building enterprise services (e.g. transactions, events, messaging, etc.). These libraries include a set of classes (called ASP.NET), which are tailored to the development of Web-based applications. ASP.NET provides an infrastructure with a set of controls that simplify both the server side (web forms that mirror the typical HTML user interface, e.g. buttons, list boxes, etc.) and client-side programming (check client capabilities and choose the appropriate interface). The ASP.NET infrastructure also includes an HTTP runtime (different from the CLR) which is an asynchronous multithreaded execution engine that processes HTTP commands. The HTTP runtime employs a pipeline of HTTP modules that route HTTP requests to a specific handler (a managed .NET class).

Short Technical Description.

- **Component types:** What most resembles a component is an **assembly**; the **manifest** is the component descriptor, it gathers in a single place all the information about an assembly: exported and imported methods and events, code, metadata and resources. Because of the programming language approach, the corresponding programming language, C♯, which looks very much like Java, includes some features of a component model: (first class) events and extensible meta data information. The compiler not only produces MSIL byte code but also generates, in the manifest, the interface description of the component (called assembly), in the form of a list of import and export types.

- **Syntactic Support in Interfaces:** The Common Type System (CTS) permits the definition and use of types across different languages. Metadata provides a uniform mechanism for storing and retrieving information about types. Together, these two facilities provide the basis of multilingual integration. Additionally, .NET provides a Common Language Specification (CLS) that describes a set of language features (e.g. primitive and composite types, natural-size types, references, exceptions) and rules for using these features (e.g. defining, creating, binding and persisting types). This specification expresses a set of naming and designing guidelines for mapping features between different languages [Cor01].

- **Extra-functional properties** .NET des not provide any support for analyzing extra-functional properties. The language enables use of meta data at run-time, which gives some possibilities for checking the properties at run-time. For example, contract-based interfaces with pre- and post-conditions may be implemented by using this feature. .NET does not provide any support for real-time applications. Further, the memory size and performance excludes it from do embedded systems domain so far.

- **Life cycle** Unlike when using traditional DLLs, the .NET model includes visibility control, which allows assemblies (and their modules) to be local to an application, and thus different DLLs with same name can run simultaneously. Further, each assembly has versioning information about itself and about the assemblies it depends on, provided either in the form of attributes in the code source or as command line switches when building the manifest. Version control is delegated to the dynamic loader, which selects the right version, local or distant, based on the assemblys version information and on a set of default rules.

Short information about .NET environments

- **Supported languages:** In contrast to the (open) OMG approach where separate formalisms (and files), are used to indicate component related information, languages and compilers being unchanged, .NET is a proprietary approach, where the program contains the information related to the relationships with other components, and the compiler is responsible for generating the information needed at execution. Current platforms include support for C♯ and Visual Basic among others.
- **Availability** .NET is used on Microsoft Windows 2000 and XP platforms. Some parts of it are ported to Windows CE, the Microsoft real-time systems. The Mono iniative (http://go-mono.com) develops an open source implementation of the .NET Development Framework. Mono includes a compiler for the C♯ language, a runtime for the Common Language Infrastructure and a set of class libraries. The runtime can be embedded into the application.

Summary of .NET

.NET is Microsoft's new paradigm for service development. It uses self-describing components (assemblies) to tackle the limitations of COM+. Each .NET assembly sets the scope for type names, and explicitly represents component dependencies. Moreover, assemblies avoid the fragmentation of disparate meta-information sources because the metadata is automatically compiled into the image PE file. Finally, .NET type information is extensible (via system attributes), can be applied to different elements (e.g. classes, methods, properties) and is available at runtime via reflection. Developers may then use these attributes that transparently integrate with the COM+ attribute-based context and transaction infrastructure.

5.2.5 CORBA and CCM

CORBA is proposed by the Object Management Group (OMG) as a standard for middleware infrastructures and a programming model for assembling and deploying distributed applications. It is part of the Object Management Architecture (OMA) [St00] which consists of

- the **CORBA bus** which maintains information about the location of components and delivers requests and responses in a standard way.
- **CORBAservices** that are predefined objects supplying functions required by most distributed applications (naming, events, security, etc.);
- **CORBAfacilities** that are object frameworks that standardize data management and user interfaces; domain interfaces that are objects for precise domains such as finance, the health industry, etc.;
- **application objects** that are objects specific to the application.

CORBA has evolved over the years as reflected in the release of three main versions of the standard. The first version simply defined a distributed object model that separated interfaces from

implementations. CORBA v1 also specified a common set of services and facilities that facilitated the development of distributed applications by integrating mechanisms for naming, event communication, life-cycle management, etc.

CORBA version 2 focused on ORB interoperability (v1 did not impose an inter-ORB protocol), and object activation management by defining the Internet Inter-ORB Protocol (IIOP) which enables interoperability across multiple ORB products, and the Portable Object Adapter (POA; see more below) which renders server objects portable and also offers various server-side configuration policies. The use of an IDL compiler combined with the runtime ORB manages cross language, cross platform and cross location independence, while the TCP/IP-based inter-ORB protocol assures cross vendor interoperability.

Finally, CORBA version 3, adopted in 2001, standardises the CORBA Component Model (CCM) which adds features and services that enable the implementation, configuration, assembly and deployment of distributed component-based applications.

The first two CORBA versions tackled interoperability through a distributed object model whereas v3 standardises a full component model. CCM increases the levels of integration and flexibility by automating tedious and error prone tasks that are usually delegated/solved by developers in ad hoc ways (e.g. deploy and install implementations, activate and configure services, life-cycle management). CCM has been designed on the basis of the accumulated experience using CORBA services.

The CCM is a server-side component model that is used to assemble and deploy multilingual components. CCM standardises and automates the component development cycle (from specification to deployment) by defining a middleware infrastructure and a set of support tools. This component model architecture permits us to define the interfaces supported by the components, automate their implementation and pack the components in assembly files (e.g. JAR, DLL) that can be automatically deployed on server hosts. The architecture uses proven design patterns [GHJV94] that enable the automation of code generation and associated usage of a container infrastructure that mediates the component access to the system services for handling security, transactions, events and persistency [Cob00]. CCM focuses on the provision of generic system services required by server applications and implemented by the container, thus freeing application code from complex and error prone tasks and allowing developers to concentrate on the business logic details. In short, the goals of CCM are very closely related to those of EJB.

Short Technical Description.

- **Component types:** these are similar to the corresponding EJB categories; i.e. session and entity categories are supported.
- **Syntactic Support in Interfaces:** A component interface is made of ports, which can be of several types
  - **facets:** named interfaces
  - **receptacles:** named connection points to other components representing external dependencies on other components' facets. (cf. required interfaces).
  - **event sources:** emit events
  - **event sinks:** consume events
  - **attributes:** named values, intended primarily for use in component configuration.
- **Other Support in Interfaces:**
- **Support for Introspection:**
- **Supported Programming languages:** CCM is a language independent model.
- **Required "Middleware"/Framework Support:** CCM, like EJB, introduces an interaction pattern based on the notion of a container. A container is automatically generated

for each component implementation and constitutes the component's view of the surrounding system [CCM]. The container shields components from the details of the underlying platform, and provides a framework (a standard runtime API) for seamless and automatic integration of core services [Cob00]. The container provides a series of uniform interfaces (called internal interfaces) that support communication with standard system services like transactions, security, persistency, and event notification). The types of internal interfaces available depends on the component category (i.e., service, session, process and entity; cf. the related EJB definitions). The container is also responsible for using callbacks to notify the component of certain events (e.g. persistency, transactions) [WSO00]. The component implements and registers these callbacks in the container to be notified of the events.

- **Extra-functional properties** CCM des not provide any support for analyzing extra-functional properties.
- **Life cycle** For each component type there is an associated 'home' component that is responsible for attributing primary keys and instantiating components. Furthermore, the component container uses an activation framework (e.g. ServantActivater, ServantLocator) that exploits the CORBA POAs mechanisms to control a components lifetime (e.g. activation, deactivation, lookup) according to a chosen policy. This way it is possible to control (depending on the component category) the activation and passivation of components, in co-operation with the persistence service, on a per-method, per-transaction, per-component (via specific callbacks) or per-container basis (this is slightly more general than the related EJB facility). Along with these lifetime policies, the CCM also standardises management policies that determine the way containers handle (on a component's behalf) transactions, security, events and persistency. The container intercepts the requests from clients and, according to the requirements (declared in the components XML configuration file), enables and executes pre-processing strategies (e.g. activation, transaction, persistence, pooling, caching) before delegating requests to the component.

  CCM supports the development process with automated mechanisms to generate and configure the necessary runtime container [CCM]. Specifically, the CCM Component Implementation Framework (CIF) defines a set of APIs and tools that automate the code generation of several management strategies (e.g. life-cycle, transaction, security, event and persistence policies). This framework automatically exposes different aspects of the implementation that may be embedded in a component's implementation [WSO00]. CCM also standardises a declarative language, called the Component Implementation Definition Language (CIDL), that is used to describe component implementations and associated persistency requirements [WSO00]. A CIF compiler reads the components CIDL description and generates default component behavior (e.g. introspection, activation, state management). The resulting implementations are called executors (e.g. facets, home, container) and provide hook methods that may be used by developers to later add custom behavior and adapt the default implementation [WSO00].

  The CIDL compiler is also responsible for generating component descriptors, which are XML files used to define the component category (e.g. entity, session), features (e.g. ports), policies (e.g. lifetime, transactions, security, events and persistency) and segmentation (i.e. delineation of independently deployable units). CCM defines several XML descriptor file-types (i.e. component descriptor, software package descriptor, assembly descriptor and property file descriptor) which conform to the WWW Consortiums Open Software Description (OSD) Data Type Definition (DTD). Component segments and descriptors are joined in a package file, i.e. a archive file that contains one or more implementations of a component and the associated XML description files. Component packages may be installed or grouped with other packages in an assembly file. Descriptor files are used at deployment-time to automatically create and configure the required POA hierarchy and to resolve component dependencies.

Short information about CCM environments

Few commercial implementations of EJB have been developed. To date, the most prominent implementation has been developed in the context of the TAO CORBA platform by the University of Washington [WSO00].

5.2.6 Real-time CORBA

Real-time CORBA is an extension to CORBA that is designed for applications with real-time requirements, such as avionics mission computing, as well as those stringent soft real-time requirements, such as telecommunication call processing. It is integrated with the CORBA 2.5 specification.

Real-time CORBA provides standard interfaces and policies that allow applications to configure and control the following system resources:

- Processor resources via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service for real-time applications with fixed priorities
- Communication resources via protocol properties and explicit bindings to server objects using priority bands and private connections
- Memory resources via buffering requests in queues and bounding the size of thread pools

CORBA and real-time CORBA have the advantage of being platform independent, in that a wide variety of programming languages support CORBA interfaces. Real-time CORBA has a particular benefit to the embedded, real-time systems market, as until recently, many such systems have had to define highly platform specific approaches to implementing many of the features proposed by the CORBA standard.

Analysis. Real-time CORBA is in itself only a standard for controlling system resources. It is up to the system designer to use the standard to configure the system to meet application requirements in a predictable way. Real-time CORBA has some shortcomings, such as not being suitable for dynamic real-time systems since it is only focused on fixed-priority based systems, and such as not addressing consistency issues.

OMGs Dynamic Scheduling proposal [OMG01a] intends is to overcome the limitations imposed by RT-CORBA in terms of dynamic scheduling.RT-CORBA only addresses static distributed systems whose resource requirements are known a priori. Offline analysis allows developers to predict the workload that will be imposed by this kind of systems. Thus, variations in these systems are bounded within a priori known operating modes. In contrast, dynamic distributed systems are those that are susceptible to experiencing unexpected dynamic changes. Therefore, in this type of system it is not feasible to predict the overall workload. This proposal introduces some modifications to the RT-CORBA specification and also provides some extensions to the standard. The proposal provides a framework that allows for the development of portable schedulers.

5.2.7 Analysis

The models described in this section represent an evolution from initial light-weight component models with support for component composition, to which support for distribution is added. Later, to support common needs in business applications, the models are extended to support an integrated container-based environment for automating the management of generic, extra-functional, properties such as transactions, security, persistency and event notifications. Only one of the models discussed, CCM, is not tied to a particular language (like Java) or operating system (like Windows). CCM was designed to align closely with the EJB specification and, apart from language independence, these component models can broadly be considered conceptual equivalents [CCM]. Both support different types of components which automatically determine the available container interfaces and the policies for managing the components state and persistency (component-managed or container-managed). Furthermore, CCM and EJB define three approaches to demarcate transactions (cf. client-managed, container-managed and server-managed) while COM+ supports only automatic transactions (MTS-managed). Moreover, COM+ defines only one type of component. This leads to a simpler programming model, but also leads to limited expressiveness and deep dependence on the MTS environment. Despite being more difficult and complex to learn

and manage, CCM and EJB architectures may be considered more flexible and open than COM+ which builds of top of the operating system services. Nevertheless, COM+ is a binary standard that allows the integration of several languages without compromising the performance.

Another significant aspect is the recurrent use of meta-information for describing the structure and behavior of components. Meta-information is widely used in CCM (e.g. the interface repository), EJB (e.g. bean descriptors) and COM+ (e.g. type libraries) but is particularly visible in .NET where the metadata is embedded in the image files and then extracted using reflection to reason about the system and control assembly, enforce security, manage serialisation, perform compiler optimisations, etc. The combination of meta-information and reflection is an interesting approach for managing type evolution.

Finally, it must be emphasised that these component models are inherently heavyweight and complex. In their present form they are not suitable for deployment in most embedded environments. Nevertheless, they exhibit many potentially interesting features that would clearly be of interest to the developers of embedded systems. What is required is research to make such features available in component model environments that are considerably more lightweight and, probably, can be tailored to specific environments on a highly configurable what you want is what you pay for basis.

Some initial work in this area has been carried out. For example, THINK ('THINK Is Not a Kernel', http://sardes.inrialpes.fr/research/think.shtml) from Inria Alpes is a minimal, low-level, component model that has been used to flexibly build software configurations at the operating system level. This develops earlier OS-level efforts such as OKKit from the University of Utah and the Spring Kernel, but adds modern notions of independent run-time deployment of components, and support for multiple interfaces. Similarly, the OpenCOM component model from Lancaster University, UK, is a lightweight component model that is being used to develop low-level programmable networking software. This model incorporates lightweight reflective mechanisms to assist in run-time deployment and dynamic reconfiguration of component compositions. Both of these component models (THINK and OpenCOM) have the potential to be applied in embedded environments, although a lot of work is required to fully validate this approach. Finally, Washington University have carried out interesting research on slimlining the CCM for application to embedded environments. This has also involved extending the CCM with support for QoS specification and validation.

5.3 Integration Platforms for Heterogeneous System Design

In this section, we review some platforms that are intended for modeling of systems, typically composed of heterogeneous components. A system design is represented as an architecture populated by interconnected components. Components can often be represented in different languages, formalisms, or even modeling paradigms.

Composition is performed at design time, typically glue code is generated automatically. The software components are wrapped with a run-time executive, which schedules the (compiled and linked) components.

A major emphasis is that the architecture description should be executable, so that simulation can be used as the major tool for design V&V. Analysis techniques can use information visible at the architectural level. Typical attributes could be period, deadline, and execution time for schedulability analysis, code size, etc.

5.3.1 MetaH

Introduction. MetaH (homepage http://www.htc.honeywell.com/metah/) is a domain-specific ADL dedicated to avionics systems which has been developed at Honeywell Labs since 1993 under the sponsorship of DARPA and the US Army. A significant set of tools (graphical editor, typing, safety, reliability, and timing/loading/schedulability analyzers, code generator...) has already been prototyped and used in the context of several experimentation projects.

In 2001, MetaH has been taken as the basis of a standardization effort aiming to defining an Avionics Architecture Description Language (AADL) standard under the authority of SAE . This emerging AADL is a domain-specific ADL developed to meet the special needs of embedded real-time systems such as avionics systems. In particular, the language can describe standard control and data flow mechanisms used in avionics systems, and important extra-functional aspects such as timing requirements, fault and error behaviors, time and space partitioning, and safety and certification properties.

Short Technical Description.
Note: MetaH in itself is only an ADL, which is under development. The rules for producing conformant component implementations are given by the current Toolset. In this description, we therefore describe MetaH together with this toolset.

- **Component types** include
  - *Macro* is a hierarchical composition of connected parts.
  - *Application* is the highest level composition, and combines a software architecture and a hardware architecture.
  - *processes*, a unit of scheduling with a protected address space in a partitioned system, and is a unit of binding to a processor. The control structure of a process must have a main outer loop, which calls the MetaH dispatcher on each iteration.
  - *packages*, and *monitors* (functioning like in Ada),
- **Visibility of Underlying Hardware** The underlying platform can be described by means of a hardware architecture. *Processors*, *Memories*, *Channels*, *Devices*. A mapping from Component types to Hardware may be provided.
- **Syntactic Support** The interface of a process or macro contains declarations of *ports*, *packages*, *monitors*, *subprograms*, *out events*, and *in events*. Components are connected by *connection declarations*, giving
  - *port connections* - message transfer between ports.
  - *event connections* - control signals, *events*, to an aperiodic process (process dispatch), or to modes (for mode switch).
  - *equivalences* - shared data and resources in terms of monitors and packages.

  Connections are strongly typed. There is no inheritance.
- **Support for Behavioral Properties** No functional properties are specified in the interface of components. A system can be described in terms of *modes* - which are run-time configurations of active processes and connections. Modes interfaces contain events. The *run-time semantics* describes how the run-time Executive works. when invoking the processes in a system.
- **Support for Timing Properties**. Components of type *process* can have (worst-case) execution times specified. This is the duration of the main outer loop on one invocation. Processes can be given periods and deadlines in a given system. There is a tool for schedulability analysis.
- **Support for Performance Properties**.
- **Support for Reliability Analysis** Components can be equipped with reliability models, which are Markov chains that relate fault events and error states. The system description must describe how errors propagate between components. A reliability analysis tool combines the reliability models of individual components into a global Markov chain, and uses a separate tool (in this case SURE/PAVE/PAWS tool from NASA Langley) or Markov chain analysis.

- **Support for Safety Analysis** Each process has its own address space in an implementation. Safety levels and memory allocation properties can be declared for components. The MetaH tool *partitioning analyzer* can partially verify that no error in a component with lower safety level can propagate to a process with higher safety level. **safety/security modeling and analysis:** The tool can check if the safety/security mechanisms provided by MetaH will enforce a specified safety/security policy. (e.g., rights of objects to access other objects.

Short information about MetaH tools, some from

http://www.rl.af.mil/tech/programs/dasada/tools.html.

- **Supported languages:** Accepts ADL specifications written in the emerging SAE standard Avionics Architecture Description Language (AADL) in both graphical and textual formats.
- **Supported languages for component implementations:** Ada, C. Many concepts are closely inspired by Ada.
- **Supported development platforms:** WindowsNT and Solaris hosts available,
- **Supported target platforms:** portable Ada95 and POSIX targets available, application source code may be written in C or Ada.
- **Status:** Core toolset fairly mature (beta-quality), reliability analysis and general system safety specification/analysis are research proof-of-concept, technologies for blended time-driven/event-driven workloads, dynamic reconfiguration, distributed hard real-time scheduling are ongoing research.
- **Availability:** Available under zero fee license; ITAR
- **Degree of Automation:** automatic production of the executable image. Can perform software/hardware allocation, and generate tailored/efficient middleware to integrate a system.
- **Analysis Support for:**
  - **Syntactic Properties:** AADL syntax/semantic checking, can translate textual to graphical and graphical to textual AADL, and check compliance of source components with AADL specifications,
  - **Functional Properties:** It is currently investigated how to (automatically) extract hybrid automata models from the generated code in order to analyze the target system.
  - **Timing Properties:** real-time schedulability modeling and analysis.
  - **Reliability Properties:** See above.
  - **Safety Properties:** See above

5.3.2 Ptolemy II

Ptolemy (http://www.ptolemy.eecs.berkeley.edu/) was initially a simulation and rapid prototyping framework for heterogeneous systems. The focus is on embedded systems, particularly those that mix technologies, including for example analog and digital electronics, hardware and software, and electronics and mechanical devices. The focus is also on systems that are complex in the sense that they mix widely different operations, such as signal processing, feedback control, sequential decision making, and user interfaces.

The Ptolemy software environment has been used for a broad range of applications including signal processing, telecommunications, parallel processing, wireless communications, network design, investment management, modeling of optical communication systems, real-time systems, and hardware/software co-design. Ptolemy software has also been used as a laboratory for signal processing and communications courses. Currently Ptolemy software has hundreds of active users at various sites worldwide in industry, academia, and government.

The first generation of Ptolemy, now called *Ptolemy Classic* is written in C++. The current version, Ptolemy II is written in Java, and produces code in Java.

Ptolemy does not really define a component model, since there is no standardized definition of component interfaces and composition at the implementation level (C or Java).

Short Technical Description.

- **Component types:** Components, called *Actors* are created in different Models of Computation (MoC). Existing MoCs include
  - CSP with synchronous rendezvous as a communication mechanism
  - CT – Continuous time, where components are described by algebraic or differential relations between inputs and outputs.
  - DE – Discrete Events. Actors communicate via events (consisting of value and time stamp). Execution of an actor is typically event-triggered. The execution semantics is realized by a discrete-event simulator, which maintains a global time-stamp-sorted queue of pending events. There is an experimental Distributed DE model of computation, using ideas from distributed DE-simulation
  - FSM – Finite State Machines, which can be used in different contexts.
  - PN – Process networks. These are Kahn process networks.
  - SDF – Synchronous Dataflow. Globally synchronous (discretely clocked) systems.
  - SR – Synchronous/Reactive. This is like the synchronous paradigm.
- **Syntactic Support:** Actors send and receive data through *Ports*. Ptolemy Classic can perform type conversions like in C. In the latest versions of Ptolemy II, there is a polymorphic type system [LX01].
- **Support for Behavioral Properties:** Each Model of Computation defines an abstract execution semantics. This can be used in simulation.
- **Support for Timing Properties**. Some Models of Computation have explicit notion of time. This can be used in simulation.
- **Support for Performance Properties**.
- **Support for Reliability Analysis:**
- **Support for Safety Analysis:**

Short information about the Ptolemy II tool

- **Supported languages:** There is a rich family of notations for graphical definitions of system structure.
- **Supported languages for component implementations:** In Ptolemy Classic, the implementation language is C++. In Ptolemy II, this has changed to Java.
- **Supported development platforms:** Windows, Linux, MacOS, X, Solaris. There is one installation that runs entirely in applets.
- **Supported target platforms:** In Ptolemy Classic, C implementations can be derived. Ptolemy Classic can generate assembly code for some programmable DSPs. Ptolemy II can generate Java code from a design.
- **Status:** Research Prototype under development.
- **Availability:** Free for download.

- **Degree of Automation:** The Java Definitions of Components is parsed, and there is support for construction of code generators.
- **Analysis Support:** Analysis support is mainly by the ability to simulate a system.
  - **Functional Properties:** In principle, Models in the FSM MoC can be parsed and used by external model checking tool. I have not found reports of significant facility.
  - **Timing Properties:** Timing properties are analysed by simulation. Some means must be devised for importing the timing properties of the actual platform. In some applications, one can use an external hardware simulator.

5.3.3 Metropolis

Metropolis (http://www.gigascale.org/metropolis/) is a research project coordinated at UC Berkeley. It is not a mature design environment; it is included here as an example of a research effort which involves a component model, where components are composed at a model level, which is at a higher level of abstraction than C or Java.

In Metropolis, an infrastructure is developed such that

1. heterogeneous components of a system can be represented uniformly, and
2. tools for formal methods can be applied naturally.

The core of the infrastructure is a *meta model of computation*, which allows one to model various communication and computation semantics in a uniform way. The meta model is defined in a variant of of timed automata. By defining different communication primitives and different ways of resolving concurrency, the user can, in effect, specify different models of computation (MoCs). The meta model is used to represent the function of a system being designed, to generate executables for simulation, and as input to formal methods built in Metropolis for both synthesis and verification in various design stages. There are stated plans to translate specifications given in many existing languages automatically to an appropriate semantics specified using the meta model

A set of coordinated tools is being developed as part of the Metropolis project.

Analysis

Metropolis goes beyond a conventional component concept, because it takes source level components and translates them into its own meta language before any simulation, verification, or code generation takes place. Thus it eliminates some of the difficultires in specifying component properties, at the expense of requiring translators from the respective source languages.

5.4 Hardware/Software Modeling Languages

In this section, we briefly mention some languages that are not component models, but can be used to model embedded systems in a modular way. The models here are mainly included to describe a part of the landscape that is adjacent to component models.

5.4.1 System C

SystemC (http://www.systemc.org/) is intended to be a standardized, highly portable technology for system-level models, an alternative to languages such as Verilog or VHDL.

Similar to HDLs, users can construct structural designs in SystemC using modules, ports, and signals. Modules can be instantiated within other modules, enabling stuctural design hierarchies

to be build. Ports and signals enable communication of data between modules, and all ports and signals are declared by the user to have a specific data type. Commonly used data types include bits, bit vectors, characters, integers, floating point numbers, vectors of integers, etc. As in VHDL, concurrent behaviors are modeled using processes.

SystemC 2.0 aims at enabling system-level modeling, i.e., modeling of systems above the RTL level of abstraction. One of the challenges in providing a system-level design language is that there are a wide range of design models of computation, design abstraction levels.

5.4.2 VHDL

VHDL is a hardware description language. It is used in a wide variety of contexts that range from complete systems like personal computers on one side to the small logical gates on their internal integrated circuits on the other side. It supports a module concept, such that abstract behavioral models may hide implementation details. The language VHDL covers the complete range of applications and can be used to model (digital) hardware in a general way.

5.5 Component Models and Integration Platforms: Summary and Conclusions

Current Trends

With respect to the evolution of different component technologies for real-time and embedded systems, we can observe the following.

- **Independent Definitions.** There are many efforts underway to define component technologies for embedded systems, often dedicated to applications in a certain domain. Examples are Koala and PECOS. These component models seem not to spread very rapidly outside the organization in which they were created. They serve the purpose of improving and software development process of their organization. Some of these models define interfaces that are not just syntactic, but include some properties that are essential for their application domain.

  An advantage of these models is that they can be tailor-made for their application domain. Disadvantages are the lack of synergy across application domains, that it is costly to develop tool support, and that such development is harder to justify for proprietary component technologies.

- **Widely Used Approaches.** Another trend is to start using more widely adopted component technologies for embedded systems. Examples are COM [LCS02] and CORBA (or its adaption to RT-CORBA). The trend is to try to avoid the cost (in terms of resources) of these technologies by using only parts that are necessary.

  An advantage is that there is already infrastructure available for these technologies, and that systems can interoperate with other system that use these technologies. A disadvantage is that these technologies do not a priori support several properties that are essential for real-time systems.

- **Lack of a Standard.** In the landscape, we have also included design tools, in which systems are designed by putting together pieces that might be termed components. Examples are MetaH and Ptolemy. The functions of these tools is more analogous to, e.g., MATLAB/Simulink. The advantage is that they support a variety of design notations. However, "components" can be assembled only in the supporting tool, meaning that different developments must all be developed in the same environment. In this perspective, these tools have similarities to tools like SCADE or UML/SDL-based tools.

- **Advanced Aspects are Still Evolving.** Many efforts are dedicated to a proper handling of extra-functional properties, including timing and QoS properties. There is a variety of developments, and no clearly identifiable "mainstream winner".

6 Findings, Synthesis, Needs

Major needs for the further development of component technology for embedded systems are the following.

- **Need for Widely Adopted Component Models and Frameworks** for embedded systems. A problem is that several application domains have application-dependent requirements on such a technology.
- **Need for light-weight implementations of Component Frameworks.** In order to support more advanced features in component-based systems, the run-time platform must provide certain services, which however must use only limited resources.
- **Incorporation of Proven Design Patterns and Idioms** In many cases, new functionality of component technology represents an incorporation and uniformization of well-understood techniques from system development. This is the case, e.g., with timing analysis in platforms like Rubus, PECOS, and MetaH, which uses well-understood schedulability analysis to perform this task. It can be expected that this trend will contribute to solving the next item on this list.
- **Uniform Specification of Rich Interfaces:** Current specification techniques for Contracts at levels 2 and up use notations and models that are quite different. It is very desirable to achieve unification and uniformization of such notations. This is not an easy task, but appears to be essential as a basis for building tool environmtns that can predict a variety of system properties.
- **Obtaining Extra-Functional Properties of Components:** Timing and performance properties are usually obtained from components by measurement, usually by means of simulation. Problems with this approach are that the results depend crucially on the environment (model) used for the measurements may not be valid in other environments, and that the results may depend on factors which cannot easily be controlled. Techniques should be developed for overcoming these problems, thereby obtaining more reliable specifications of component properties.
- **Platform and Vendor independence:** Many current component technologies are rather tighly bound to a particular platform (either run-time platform or design platform). This means that components only make sense in the context of a particular platform. The conclusion is that either we will see a future dominance of one platform (e.g., as MatLab/Simulink) for each application domain, or that platform-independent models must be developed.
- **Efforts to Predict System Properties:** The analysis of many global properties from component properties are hindered by inherent complexity issues. Efforts should be directed to finding techniques for coping with this complexity, including to find architectures and design principles that lower the complexity of predicting system properties.
- **Addressing Timing Properties:** The analysis of timing properties face the problem of being dependent on the underlying platform. In the short term, it does not appear feasible to find a technique for specifying timing properties in a platform-independent way, but rather techniques for living with platform-dependence must be devised.
- **Component Certification:** In order to transfer components across organizations, techniques and procedures should be developed for ensuring the trustworthiness of components.
- **Component Adaptation:** In order to meed the resource limitations of embedded systems, there is a desire to be able to adapt components to use exactly the resources and services that are need in a particular service. We have not found reports of significant techniques for achieving this.

- **Component Noninterference:** Particularly in safety-critical applications, there is a need to ensure separation and protection between component implementations, in terms of memory protection, resource usage, etc. There are techniques for achieving this, but it is desirable to turn such techniques into tools that can reliably and automatically ensure noninterference.
- **Tool support:** The adoption of component technology depends on the development of tool support.

7 Missing pieces

Technical problems that are not yet overcome are the following

- In the landscape, we have observed the two seemingly conflicting trends of developing proprietary component models with rich interfaces, and of using (parts of) widely adopted component technologies. Might it be possible to unify these trends by developing component technologies with rich interfaces that are compatible with CORBA or .NET? (e.g., they could use the syntactic standards imposed by CORBA or .NET, and add important functional/extra-functional properties)
- Component verification and certification are so far unsolved problems. There seem to be no standardized procedures for ensuring component trustworthyness.
- There are several suggestions for how to handle functional and extra-functional properties of system design (timing, QoS, etc.). Widely accepted techniques for specifying functional and extra-functional properties of components and of middleware remain to be developed. (see further in Section 4)
- There is a clearly perceived need to be able to predict system properties from properties expressible in component interfaces. Such properties could include timing, QoS, memory and power consumption. A major motivation is to support system integration. Practical, widely adoptable solutions remain to be developed. (see further in Section 4)

8 Clearly-Identified Priorities

In this section, we discuss prioritized areas for further work.

- *Predicting System Properties.* A research challenge today is to predict system properties from the component properties. This is interesting for system integration, to achieve predictability, etc. Some kye obstacles to overcome in order to address this challenge are as follows.
  - Predicting system properties from component properties is often computationally expensive. A possible remedy is to devise design principles for architectures and components that yield tractable analysis procedures for interesting classes of properties.
  - Extra-functional properties of components are hard to obtain in a reliable way. They can be obtained by measurement (e.g., in simulations), but care must be taken for generalizing the measured properties across varying environment and platform conditions. For safety-critical systems, static analysis of execution time can be developed further.
- *Development of Widely Adopted Component Models for Real-Time Systems.* From the report, it is obvious that such a model should support key extra-functional properties, including timing and QoS. Such a model should be supported by technology for generating necessary runtime infrastructure (which must be light-weight), generation of monitors to check conformance with contracts, etc. The trend towards open an integrated systems implies that it should be possible for a system to use both a component model specific for real-time systems, and some of the widely used component technologies.

9 Standardization Efforts

Corresponds to Deliverable 4.1.2:

This section provides an overview of standards that are relevant to the ARTIST components working group. It is broadly split into two main sections: specification standards and implementation technology standards. The aim is to distinguish between standards for specification and modeling, such as UML, that define modeling concepts related to real-time modeling and components, and implementation standards, which are focused at realizing these concepts at the implementation level.

9.1 Specification Standards

Over the last decade, there has been an increasing emphasis on the development of modeling and specification languages appropriate for describing software engineering concepts. The Unified Modeling Language (UML) was one of the first modeling languages to standardize software engineering concepts, and it and related OMG standards are currently the de-facto route for incorporating new concepts into the software industry. This section briefly introduces some of the key standards and describes work currently being done to incorporate real-time components into existing standards such as UML.

9.1.1 UML 2.0

The Unified Modeling Language (UML) is now the de-facto industry language for specifying and designing software systems. Since its inception in 1997, the scope of the language has become ever wider. UML now provides support for a wide variety of modeling domains, including real-time system modeling.

Unfortunately, the success of UML has come at a cost, resulting in a bloated and complex language, as new modeling concepts have been repeatedly “mud-packed” into the definition. Furthermore, the specification of the language (a meta-model of its abstract syntax with weakly defined semantics) has also become difficult to manage and hard to understand due to its size and complexity.

The UML 2.0 effort is an attempt by the OMG to address these shortcomings. The aim is that UML should become a family of languages, each based on a common semantic core. Thus, specific variants of UML will be defined for specific application areas: e-business, real-time, systems engineering, warehouse meta-data and so on. Another important aim is that UML 2.0 should be defined more precisely in order to facilitate the more rigorous use of the language.

Currently, a number of consortia are submitting a variety of proposals to the OMG for the revised standard. The work has been split into four main areas: infrastructure (the core modeling concepts supported by UML), superstructure (the modeling concepts that are used to model specific views of a system, e.g., state behavior), OCL (the object constraint language that supports the semi-formal specification of constraints) and diagram interchange (tool interchange of diagrammatical syntax). The intention is that the infrastructure model will define the semantics for the core concepts used by UML. The superstructure will then be defined in terms of this core, thus providing a firmer interpretation of the UML language as a whole.

The task involved in refactoring UML as a family of languages is not straightforward. Much work needs to be done to define an infrastructure that will successively support the definition of a wide spectrum of languages. Furthermore, first class extension mechanisms are required to support the incremental extension and combination of language components to create new languages. Finally, defining a semantics for these language components is a significant challenge in itself - necessitating that UML has a well defined semantic domain and appropriate semantic mappings.

9.1.2 Components at the Model Level

Components being the now most widespread structuring entities at implementation level (seen as executables, binaries or library elements), the component paradigm tends also to play such role at the modeling stage. One of the motivations is related to the difficulty to have reusable components having totally known and mastered dynamics, in particular, on RT and concurrency aspects. Incoming component-based approaches ([DW99, HS99, ABM00, GPJ02]) tend to use components as a higher-level modeling artefact that may be used whatever the nature of the model (specification design, implementation, ...) and derived throughout the system development till implementation. Implementation part of a component becomes one of its aspects only relevant for the implementation stage.

The evolution of this concept from UML1.x to UML 2.0 confirms this tendency. Components in UML 2.0 are likely to get a more extensive treatment than in previous versions of UML. Considered as a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces in the UML1.x, components become more abstract structuring entities in UML 2.0. They will be defined in the superstructure of UML 2.0.

UML 2.0 components are then a modular part of a system that may be modelled and refined throughout the development life cycle. A component is viewed as an autonomous unit within a system or subsystem. It has one or more ports, and its internals are hidden and inaccessible other than as provided by its interfaces. As a result, the aim is that components and subsystems can be flexibly reused and replaced by connecting ("wiring") them together via their provided and required interfaces. Components also support reuse through an import mechanism, which permits the elements of a component to be imported into another component. UML 2.0 component model is very close to the main frame of the component model described in previous section 2 allowing thus to use very easily UML as modeling language or ADL to support a CBSE methodology.

Although, this approach is not yet mature, at least due to the introduction of the concept only in the incoming version of UML standard, some proposals already introduce this notion in relation with real-time preoccupation through attaching RT QoS to the component interfaces [GPJ02]. In this context, component composition issue at design stage becomes a question of QoS composition among the component models. This raised a strong interest on MDA techniques that facilitate: model weaving (http://www.qccs.org/) for the component composition; and the mapping and transformation of abstract models into detailed models for the implementation synthesis [GTT02].

9.1.3 Real-time UML Profile

The UML contains in native some capabilities to support real-time aspects: either for qualitative aspects such as concurrency (Active objects, concurrent states, ...), or for quantitative aspects such as time event. Nevertheless, these real-time features of the UML are inadequate. For that reason, OMG has initiated a work dedicated to define a UML profile specific to real-time systems development.

The real-time UML profile [OMG01b] (actually the profile for Schedulability, Performance, and Time Specification) defines standard paradigms of use for modeling of time-, schedulability-, and performance-related aspects of real-time systems. The intention is to

- enable the construction of models that could be used to make quantitative predictions regarding these characteristics,
- facilitate communication of design intent between developers in a standard way,
- enable inter operability between various analysis and design tools.

To support this, the specification defines (as a meta-model) a complete, but generic model of some of the key concepts association with scheduling, performance modeling and times events. Main concepts introduced in the SPT profile are Quality-of-Service (in short QoS) and Resource. From

these concepts, it then defines in specific sub-profiles more adequate concepts for performance and schedulability analysis. Thus it includes models of the semantics and mappings to common real-time middleware standards such as real-time CORBA. The final version of this standard should be delivered for summer 2002. The main issue regarding to this standard is that it is relative to UML1.x, and that it should have to be revised for incoming UML2.x

The QoS concept introduced in the SPT profile was mainly an abstract concept. So following the SPT profile initiative, the OMG has started another reflection soliciting proposals for a UML profile that defines standard paradigms of use for modeling QoS and fault tolerance aspects of real-time systems. To achieve this goal, the following RFP has then been launched: UML profile for Modeling QoS and Fault Tolerance Characteristics and Mechanism (ad/2002-01-07).

9.1.4 EDOC

EDOC is a UML Profile for Enterprise Distributed Object Computing [OMG01c]. It describes how UML concepts can be used to model system component structures and behavior through the definition of the Component Collaboration Architecture (in short CCA). A component that meets CCA criteria is known as a ProcessComponent. Such components can be subsequently implemented by physical components using special technology.

The architecture of this profile is broken down into various levels and permits description of both same level component interactions and exchanges between different level components.

A CCA component can define the properties required to configure it for use. These components intercommunicate through ports, which are in turn linked together by connections. Each port can be transactional and synchronous. A port also has a direction: Initiates or Responds. An initiator port sends a message to begin a conversation. A responder port answers this message and may, if appropriate, pursue the conversation.

A port is further broken down into specialized FlowPort, ProtocolPort and OperationPort concepts. The simplest of these is the FlowPort, which defines entering or exiting data flows. An OperationPort contains a set of FlowPorts representative of typical operation call/return behavior. Only one of the FlowPorts acts as initiator and represents the operation call. The others act as responders and provide operation return values.

A ProtocolPort defines the use of a protocol for complex communications between components. A protocol defines the type of conversation held between an Initiator and a Responder. This definition states which messages are sent and received but also defines their choreography as modeled by an activity graph. A protocol is made up of subports that may themselves be ProtocolPorts but in fact break down, at the finest level, into FlowPorts.

An interface is a special protocol case, with all the usual possibilities of an object interface. It can therefore only contain OperationPorts and FlowPorts.

To be executable, a ProcessComponent must have a "technology profile". This profile provides specifications for automatic translation of CCA components into implementation models (Java, CORBA, etc.).

CCA can only be expressed using basic UML 1.4 notation. A few extensions and additional notations have, however, been defined to enhance readability and compactness for those tools that support CCA.

9.1.5 MDA

The Model Driven Architecture (MDA) is the OMG's new flagship architecture that aims to integrate its (and other) standards within a model driven approach to system development ([StOG00, MM01, StOG01]). MDA encapsulates many important ideas - most notably the notion that real benefits can be obtained by using modeling languages to integrate the huge diversity of languages used in the development of systems.

In MDA, modeling languages are categorized as being platform independent (i.e. specification oriented) and platform independent (i.e. implementation oriented). Note that a modeling language can be a language at any level of abstraction. Examples of platform independent languages include UML itself (when used for specification). Middleware standards such as CORBA and programming level languages (e.g. Java Beans) are examples of platform specific languages.

Mappings (a key component of MDA) define the relationships between these languages. By abstracting away from platform specific details, the intention is that system development is driven through platform independent models that can be semi-automatically translated into any platform specific language for which a standard mapping has been defined. Thus, platform independence is obtained along with greater flexibility in deployment - if a new technology emerges (e.g. .Net), then all that is required is to apply a new set of mappings.

Platform independent and platform specific mappings are good examples of vertical transformations. However, MDA goes potentially far beyond this. For example, horizontal mappings between platform specific languages may be defined as a means of integrated different modeling perspectives at the specification level (e.g. process models, system artefacts, software specifications). In short, MDA offers a framework that has the potential to model and integrate all aspects of system development.

Many vendors are already claiming support for MDA (e.g. a code generator could be viewed as a mapping tool!). However, in practice there are significant issues to be addressed. Mechanisms must be defined that support executable, but declarative mappings between languages. The semantics of these languages must be defined well enough to ensure that mappings are semantic preserving, whilst more powerful extension mechanisms are required to support the reuse of mappings. These and many other issues are all currently being debated within the OMG (see www.omg.org/mda).

9.1.6 MDA & current industrial Real-time-UML tools

In the real-time application area, this model-oriented trend is also very active and promising. Currently, there are four main model-oriented industrial approaches supported by tools: UML-RT used with Rose-RT, ROPES with Rhapsody, ARTiSAN and UML/SDL with the Tau UML/SDL suite.

Within UML-RT, an application is seen as a set of entities called "capsules"' which support logical concurrency. These capsules have a state machine as behavior specification and may exchange signals to communicate. Models built in this way are said to be executable, meaning that at any moment in design, it is possible to produce an executable application matching the UML model. In this case, the mapping is achieved via code generation.

For ROPES and ARTiSAN approaches, real-time application modeling is a 3-stage process: i) building a "functional"' model with class and state diagrams; ii) building a specific tasking model with class diagrams containing only active objects (execution tasks); iii) describing the mappings between the two models. The main drawback of this "family" of methods is that it requires advanced real-time development skills to build the tasking model and map it with the "functional" model. While there are some "shortcuts" available ([AKZ97, p. 482]) to facilitate this activity, no transformation rules are provided as could be done within a fully MDA-based approach.

The approach proposed by Telelogic is based on the use of both UML and SDL languages. It consists of building UML models at the analysis stages using active objects as concurrency supports and SDL within design-time. Reference document [IT99] defines modeling rules for mapping a UML-oriented model into an SDL-oriented model. When SDL models are finished, the engineer may generate code to produce an executable application.

All these methodologies may be considered as MDA-based approaches for mainly two reasons. Firstly, they clearly promote the model paradigm to develop applications; and secondly, they provide code generation taking into account structural and behavior specifications for model mapping to implementation languages such as C, C++, JAVA, ... Nevertheless, they do not exploit all the potentialities of MDA. Their application models are often only PSM-like for "executable" reasons.

For modeling purposes, the user is thus led very quickly to resort, for an executable model, to a programming language such as C++, . Although action semantics have been standardized by OMG [OMG02] there are still only a few tools that have integrated this feature, which allows building of executable models independently of any programming language.

While these approaches are usually based on a several stage process, they do not provide the refinement mapping rules that could facilitate application development and, above all, be highly useful in promoting seamless development processes. Finally, the existing UML-based methods for real-time applications still require considerable knowledge of real-time software technology (and the different programming models promoted by these tools) to develop real-time systems.

9.1.7 Towards MDA components?

MDA approach has given rise to a particular interest of the RT community (e.g., the first edition of the Summer School on MDA for embedded systems held in Brest, Sept. 2002, http://sancy.ensieta.fr/mda/) [Dou02]. However, this subject remains largely open, in particular, to identify and structure the various artefacts related to MDA, such as: dedicated metamodels (e.g., for business domain and technical domain), specific target models (also called PSMs Platform Specific Models), transformation procedures, model weaving, mapping and transformation rules, in particular, concerning RT QoS, but also for implementation synthesis, test generation, proof synthesis (see for example Josephil Challenge launched for the second edition of the summer school: http://sancy.ensieta.fr/mda/JOSEFIL/).

Incoming MDA-based workbenches will consist of various parts that may interoperate:

- (i) Documents (method book, guide lines, user guides,); ¿(ii) Profiles (SPT, SPEM, EDOC,); ¿(iii) Tools (UML modeler, Code generator, Model transformer, Model validator,).

Moreover, these MDA parts may be plug on a bus in order they interoperate. For example, the ECLIPSE initiative (www.eclipse.org) provides a specific plug-in, EMF (Eclipse Modeling Framework), ensuring the construction of UML-based MDA plug-ins of ECLIPSE.

Even if it is not well defined today, it seems logical that components will also play this structuring role [BG02]. And near future should give rise to "MDA-Components" whose nature could be clarified thanks standard stereotypes such as Tool, UML Profile, Document,

In fact, currently not a lot of things have been set related to MDA and it still remains a lot of work to do to clarify MDA and its related concepts. In particular, as CBSE methodology have been developed to support more efficient the use of the component artefact, Model Driven Engineering methodologies have to be define to exploit the all the potentialities of the MDA technologies and concepts.

9.2 Implementation Technology Standards

The majority of implementation standards relevant to components tend to focus on middleware, i.e. the communication and interface aspects of components. Each of these standards emphasizes the important of independence from the technology use to implement the internal functionality of components. Because of this, we will not discuss the plethora of programming languages, etc, that can be used to implement components in this section.

CORBA is described in Section 5.2.5.

9.2.1 SOAP

SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. As such, SOAP can be seen as an important standard for interchanging date between distributed components. SOAP does not itself provide implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules. This allows SOAP to be used in a large variety of systems ranging from messaging systems to RPC.

SOAP consists of three parts:

- The SOAP envelope, which defines an overall framework for expressing what is in a message;
- The SOAP encoding rules defines a serialization mechanism that can be used to exchange instances of application-defined datatypes. These may be simple or structured datatypes.
- The SOAP RPC representation defines a convention that can be used to represent remote procedure calls and responses.

Just as with CORBA, the key advantage of SOAP is that it is platform independent and is not tied to any implementation specific messaging mechanism or software architecture.

9.3 Conclusions and Challenges

All the above standards are relevant to the ARTIST component working group as they each attempt to standardize a variety of aspects of real-time and component based design in isolation. As a result, there are many opportunities for additional work to unify both the real-time and component perspectives and also to provide a stronger foundation for their definition and deployment. These include the following:

- Integration of Real-time and Embedded QoS within UML2 component model: Currently, little consideration has been given to the expression of real-time and QoS aspects in UML component models. Such an approach would require the definition of additional notational facilities to facilitate the capture of these aspects, along with a definition of their semantics.
- Traceability management/control of Real-time QoS of a component all along the development process: By providing a model of change management/control it should be possible to provide support for the management of components throughout its lifetime (an essential requirement for change management and upgrades). Such a facility could potentially be based on an extension to emerging process management modeling languages being developed in the industry such as SPEM.
- Definition of Performance/schedulabilty analysis methodology well-suited for such MDA component-based approaches: It is clear that the deployment of components within an MDA life cycle could be developed whereby components could be specified in a platform independent way, and then mapped to various component technologies. In order to achieve this, significant work needs to be done to develop models of platform specific component languages and to define rules for mapping from platform independent components to platform specific models or to middleware standards such as CORBA and SOAP. These mappings must be shown to be correct with respect to certain semantic preserving properties, including QoS.
- Link between extra-functional engineering requirements and Real-time/Embedded QoS of UML-based models: By modeling extra-functional properties of real-time systems, it should be possible to build rich, component based modeling languages that capture a variety of system engineering perspectives. This would tie in nicely with work going on in the OMG to define a UML profile for systems engineering.

Glossary (for the Component Part)

What is a component? In **classic engineering disciplines** a component is a self-contained subsystem that can be used as a building block in the design of a larger system. The component provides the specified service to its environment across well-specified component interfaces. An example of such a component is an engine in an automobile, or the heating furnace in a home. The component can have a complex internal structure that is neither visible, nor of concern, to the user of the component. An ideal component should maintain its encapsulation when used in a larger context. The larger system should be constructed from nearly autonomous components that can be integrated without violating the principle of composability: that properties that have been established at the component level will also hold at the system level.

What is a component in **software engineering**?
That's a question of point of view. According to [BBB$^+$00], advocates of software reuse equate components to anything that can be reused; practitioners using commercial off-the-shelf (COTS) software equate components to COTS products; software methodologists equate components with units of project and configuration management; and software architects equate components with design abstractions.

The best accepted definition in the software industry world is based on Szyperski's work [Szy98]:
a component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition.

In this report, we follow the definition by Szyperski, and in particular stress the separation between component implementation and component interface. We will view a component as a software implementation that can be executed on a physical or logical device. The component implements one or more interfaces. Idealy, there should be no context dependencies that are not captured by the component interface. This last sentence must be applied with some common sense, because in practice most interfaces capture only certain aspects of a component's behavior.

Component Model A component model prescribes how components interact with each other and with the component framework, in order that independently developed component can be deployed in the composition environment. The model thereby specifying the standards and conventions imposed on developers of components. A component model states what it means for a component to implement a given interface; it imposes constraints on components so that they can be located, communicate using agreed protocols, etc.

Component Framework A component framework provides a variety of services to support and enforce the component model. In many respects component frameworks are like special-purpose operating systems. Using an analogy, components are to frameworks what processes are to operating systems. The framework manages resources shared by components, and provides the underlying mechanisms that enable interaction between components. Note that it is not necessary that the framework have a runtime existence independent of components. For example, the framework can be a run-time executive that is bundled with a component system during compilation.
A few authors use the term "component framework" in a wider sense, to encompass both "framework" and "component model".

Interface An **interface** is a specification of an access point for a component. A component can exhibit several interfaces. The interface specifies its externally visible features, which are both necessary on one hand for the user of a component and on the other hand for an implementer. Ideally, there should exist no communication between a component and its environment that is not specified in its interface; this statement must be used with care, however, since an interface often specifies only syntactic properties, and often ignores sharing

of resources, timing properties, etc. An interface that also specifies behavioral or extra-functional properties is called a **rich interface.** We note that our definition of interface (for components) is more general than the meaning of interface, e.g., in OO programming, where an interface is simply the signature of a collection of methods.

Contracts A **contract** is a specification of obligations of a component, in terms of properties that can be observed in its interface. Contracts ensure that independently developed components obey certain rules so that components interact (or can not interact) in predictable ways, and can be deployed into standard build-time and run-time environments. An often cited classification of contracts is given in [BJP99], where 4 levels of contracts are defined: syntactic (conformance of signature), behavioral (expressed e.g. by pre- and postconditions), synchronization, and quality of service (response time and so on). The component technologies mostly used today (COM, EJB, .NET) provide support only for contracts at the syntactic level.

Functional Properties: properties relating to the data handled by a system influencing its functionality. This includes protocol aspects that relate to the order in which data are exchanged.

Extra-Functional Properties (or non-functional properties): properties not directly influencing the functionality of a system, but rather its "quality". Example of such extra-functional properties concern response-time, memory-use, energy consumption, etc. In the domain of time related properties, the distinction between functional and extra-functional properties is not always clearly defined, as the system behavior is often explicitly time dependent, and timely response may be part of the functionality of a system.

Introspection Introspection is the ability of a system to monitor itself, e.g., to answer queries about its interfac(es) or to monitor its extra-functional properties.

Middleware

Port: a port defines an interaction point between a component instance and its environment. The terms "port" and "connector" originate in architecture description languages.

Connector: a connector connects 2 or more ports and restrict the possible interactions between different objects of a systems to those defined by its

What is an integration environment ?

References

[ABM00] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development : The KobrA approach. In *Proc. 1st International Software Product Line Conference (SPLC-1)*, pages 289–309, Denver, Colorado, Aug. 2000.

[Abr96] J.-R. Abrial. *The B book - Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AFM+02] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: A tool for modeling and implementation of embedded systems. In P. Stevens J.-P. Katoen, editor, *8th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Grenoble, France*, LNCS 2280, pages 460–464, 2002.

[AKZ97] M. Awad, J. Kuusela, and J Ziegler. *Object-Oriented Technology for Real-time Systems: A Practical Approach Unisg OMT and Fusion.* Prentice Hall, 1997.

[BBB+00] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering, Volume II. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie-Mellon University, May 2000.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice.* Addison Wesley, 1998.

[BCP+01] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos. a tool for verifying real-time properties of embedded systems. In *Conference on Decision and Control, CDC'01, Orlando*. IEEE, Nov. 2001.

[BG02] J. Bézivin and S. Gérard. A preliminary identification of MDA components, 2002. Position Paper, OOPSLA 2002 Workshop: Generative Techniques in the context of Model Driven Architecture.

[BGM02] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A validation environment for component-based real-time systems. In E. Brinksma and K.G. Larsen, editors, *Proc. 14th Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 343–348, Copenhagen, Denmark, 2002. Springer Verlag.

[BGS00] S. Bornot, G. Gössler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS 2000*, volume 1785 of *LNCS*, pages 109–126. Springer-Verlag, 2000.

[BJP99] A. Beugnard, J.-M. Jézq́uel, and N. Plouzeau. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

[BK98] N. Brown and C. Kindel. Distributed component object model protocol – dcom/1.0. Internet-draft, IETF, Jan. 1998.

[BLP+02] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Y. Watanabe, and G. Yang. Concurrent execution semantics and sequential simulation algorithms for the metropolis meta-model. In *Proc. CODES'02, 10th Int. Symp. on Hardware/Software Co-Design*, Estes Park, Colorado, April 2002.

[BMW89] H. Beilner, J. Mater, and N. Weissenberg. Towards a performance modelling environment: News on HIT. In *Proc. Int. Conf. Modeling Techniques and Tools for Computer Performance Evaluation*, pages 57–75. Plenum Press, 1989.

[Box00] D. Box. House of COM: Is COM dead? *MSDN Magazine*, Dec. 2000.

[Bro96] K. Brockschmidt. What OLE is really about, 1996.

[Cas95] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[CCM] The CORBA & CORBA Component Model (CCM) page. http://www.ditec.um.es/ dsevilla/ccm.

[CdAH+02] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Jurdzinski, and F.Y.C. Mang. Interface compatibility checking for software modules. In E. Brinksma and K.G. Larsen, editors, *Proc. 14th Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 428–441, Copenhagen, Denmark, 2002.

[Chi98] G. Chiola. Petri nets versus queueing networks: similarities and differences. In M. Silva and G. Balbo, editors, *Performance Models for Discrete Event Systems with Synchronisations: Formalisms and Analysis Techniques*, pages 121–134. KRONOS, Zaragoza, Spain, 1998.

[CHK99] M. Dal Cin, G. Huszerl, and K. Kosmidis. Evaluation of safety-critical systems based on guarded statecharts. In R. Paul and C. Meadows, editors, *Proc. of the Fourth IEEE International Symposium on High Assurance Systems Engineering*, pages 37–45. IEEE, 1999.

[CL02] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems.* ArtechHouse, 2002.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, Massachusetts, 1988.

[Cob00] E. Cobb. CORBA Components: The industry's first multi-language component standard, June 2000. OMG meeting tutorial available at http://www.omg.org/cgi-bin/doc?omg/00-06-01.

[Con02] C. Constantinescu. Impact of deep submicron technology on dependability of VLSI circuits. In *Proc. Dependable Systems and Networks (DSN'02)*, pages 205–214. IEEE, June 2002.

[Cor95] Microsoft Corporation. The component object model specification, Oct. 1995. 24 ed.

[Cor01] Microsoft Corporation. .NET framework developer's guide, 2001. http://msdn.microsoft.com/library/default.asp.

[CW02] S. Clarke and R.J. Walker. Towards a standard design language for AOSD. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 113–119. ACM Press, 2002.

[DH01] W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. *Journal of Formal Methods in System Design*, 19(1):45–80, 2001.

[DHMC96] M. Diefenbruch, J. Hintelmann, and B. Mller-Clostermann. The QUEST-approach for the performance evaluation of SDL-systems. In R. Gotzhein and J. Bredereke, editors, *Proc. FORTE '96*, pages 229–244. Kluwer, 1996.

[DKB98] P. D'Argenio, J. Katoen, and E. Brinksma. An algebraic approach to the specification of stochastic systems, 1998.

[Dou02] B.P. Douglass. Model driven architecture and Rhapsody. Technical report, I-Logix, 2002.

[DW99] D. F. D'Souza and A. C. Wills. *Objects, components, and frameworks with UML : the catalysis approach.* ACM Press and Addison-Wesley, Reading, Mass., 1999.

[ESC] ESC Java webpage: http://research.compaq.com/SRC/esc/.

[ESCW01] H. El-Sayed, D. Cameron, and M. Woodside. Automation support for software performance engineering. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):301–311, 2001.

[FEHC02] A.V. Fioukov, E.M. Eskenazi, D.K. Hammer, and M.R.V. Chaudron. Evaluation of static properties for component-based architectures. In *Proc. 28th Euromicro Conf.*, pages 33–39, Dortmund, Germany, Sept. 2002. IEEE Computer Society Press.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H.Theiling, St. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In T.A. Henzinger and C.M. Kirsch, editors, *Proc. EMSOFT 2001*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, Tahoe City, CA, Oct. 2001. Springer Verlag.

[FK98] S. Frolund and J. Koistinen. Quality-of-Service specifications in distributed object systems. *Distributed Systems Engineering*, 5(4):179–202, Dec. 1998.

[FW98] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In F. Mueller and A. Bestavros, editors, *ACM SIGPLAN Workshop LCTES'98*, volume 1474 of *Lecture Notes in Computer Science*, pages 16–30, Montreal, Canada, June 1998. Springer Verlag.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, June 1995.

[GHG+93] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science series. Springer Verlag, 1993. This is currently out of print, but a (large) postscript file for the entire book can be obtained from the following URL http://www.sds.lcs.mit.edu/spd/larch/pub/larchBook.ps.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., July 1994.

[GLM00] S. Gnesi, D. Latella, and M. Massink. A stochastic extension of a behavioral subset of UML statechart diagrams. In *Proc. HASE: 5th Int. Symp. on High-Assurance Systems Engineering*, pages 55–64. IEEE Computer Society Press, 2000.

[Gos95] C. Goswell. The COM programmer's cookbook, Sept. 1995.

[Gös01] G. Gössler. Prometheus — a compositional modeling tool for real-time systems. In P. Pettersson and S. Yovine, editors, *Proc. Workshop RT-TOOLS 2001.* Technical report 2001-014, Uppsala University, Department of Information Technology, 2001.

[GPJ02] S. Gérard, P. Petterson, and B. Josko. Methodology for developing real time embedded systems. Pub IST-1999-10069, CEE, Paris, France, 2002.

[GS02] G. Gössler and J. Sifakis. Composition for component-based design. In *International Symposium on Formal Methods for Components, Objects and their Implementation, FMCO'02*, LNCS, 2002. invited presentation.

[GTT02] A. Gŕard, F. Terrier, and Y. Tanguy. Using the model paradigm for real-time systems develoment: ACCORD/UML. In J.-M. Bruel and Z. Bellahsène, editors, *Proc. OOIS 2002 Workshops, Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 260–269, Montpellier, France, 2002. Springer Verlag.

[Hen01] T.A. Henzinger. Giotto: A time-triggered language for embedded programming. In T.A. Henzinger and C.M. Kirsch, editors, *Proc. EMSOFT 2001*, volume 2211 of *Lecture Notes in Computer Science*, pages 166–184, Tahoe City, CA, Oct. 2001. Springer Verlag.

[Her02] H. Hermanns. *Interactive Markov Chains and The Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science.* Springer Verlag, 2002.

[Hil96] J. Hillston. *A Compositional Approach to Performance Modeling.* Cambridge University Press, 1996.

[HJPN02] W.M. Ho, J.-M. Jzquel, F. Pennaneac'h, and N.Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of 1st ACM International Conference on Aspect Oriented Software Development, AOSD 2002*, Enschede, The Netherlands, April 2002.

[HS99] P. Herzum and O. Sims. *Business Component Factory: A Comprehensive Overview of Component Based Development for the Enterprise.* John Wiley and Sons, 1999.

[IEC95] IEC. Application and implementation of IEC 61131-3. Technical report, IEC, Geneva, 1995.

[Ilo] Ilogix rhapsody. http://www.ilogix.com.

[IN02] D. Isovic and C. Norström. Components in real-time systems. In *Proc. RTCSA 2002, 8th International Conference on Real-Time Computing Systems and Applications*, Tokyo, Japan, March 2002.

[IT99] ITU-T. Recommendation Z.109 : Languages for telecommunications applications - SDL combined with UML. Technical report, Geneva, Nov. 1999.

[IT00] ITU-T. Recommendation Z.120. Message Sequence Charts. Technical Report Z-120, International Telecommunication Union – Standardization Sector, Genève, 2000.

[JF00] H. Jubin and J. Friedrichs. *Enterprise JavaBeans by Example.* Prentice Hall, New Jersey, 2000.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[Ker01] L. Kerber. Scenario-based performance evaluation of SDL/MSC-specified systems. In R. Dumke, C. Rautenstrauch, A. Schmietendorf, and A. Scholz, editors, *Performance Engineering – State of the Art and Current Trends*, volume 2047 of *Lecture Notes in Computer Science*, pages 185–201. Springer Verlag, 2001.

[Kle75] L. Kleinrock. *Queueing systems – Volume 2: Theory.* John Wiley and Sons, 1975.

[Kle76] L. Kleinrock. *Queueing systems – Volume 2: Computer Applications.* John Wiley and Sons, 1976.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[KS03] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proc. 6th IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Hokkaido, Japan, May 2003.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[LB99] G.T. Leavens and A.L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. Woodcock and J. Davies, editors, *FM'99 – Formal Methods: World Congress on Formal Methods in Development of Computer Systems, Toulouse, France*, LNCS 1709, 1999. http://www.cs.iastate.edu/ leavens/JML.html.

[LCS02] F. Lüders, I. Crnkovic, and A. Sjögren. Case study: Componentization of an industrial control system. In *Proc. 26th Annual International Computer Software and Applications Conference - COMPSAC 2002*, Oxford, UK, Aug. 2002. IEEE Computer Society Press.

[Lee01] E.A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 2001.

[Lew98] R.W. Lewis. *Programming industrial control systems using IEC 1131-3.* IEE, 1998.

[Loo] Loop Project webpage: http://www.cs.kun.nl/ bart/LOOP/.

[LX01] E.A. Lee and Y. Xiong. System-level types for component-based design. In T.A. Henzinger and C.M. Kirsch, editors, *Proc. EMSOFT 2001*, volume 2211 of *Lecture Notes in Computer Science*, pages 237–253, Tahoe City, CA, Oct. 2001. Springer Verlag.

[Mau96] S. Mauw. The formalization of message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1643–1657, 1996.

[MCB84] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalised stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. on Computer Systems*, 2(2):93–122, May 1984.

[Mey91] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[MG02] E. Meijer and J. Gough. Technical overview of the Common Language Runtime, 2002. White paper.

[Mic97] Sun Microsystems. Javabeans, July 1997. http://java.sun.com/products/javabeans/docs/spec.html.

[Mic02] Sun Microsystems. Enterprise JavaBeans specification, 2002. http://java.sun.com/products/ejb/index.html.

[MM01] J. Miller and J. Mukerji. Model driven architecture (MDA), July 9 2001. OMG, Draft Specification ormsc/2001-07-01.

[Mol82] M.K. Molloy. Performance analysis using Stochastic Petri Nets. *IEEE Trans. on Computers*, C-31(9):913–917, Sept. 1982.

[MSP+00] M. Morisio, C. B. Seaman, A. T. Parra, V. R. Basili, S. E. Kraft, and S. E. Condon. Investigating and improving a COTS-based software development. In *Proceedings of the 22nd international conference on Software engineering*, pages 32–41. ACM Press, 2000.

[NAD+02] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P.O. Müller, C. Zeidler, T. Genssler, and Reinier van den Born. A component model for field devices. In J. Bishop, editor, *Component Deployment, IFIP/ACM Working Conference*, volume 2370 of *Lecture Notes in Computer Science*, pages 200–209. Springer Verlag, June 2002.

[Nee91] S. Neema. *System-Level Synthesis of Adaptive Computing Systems.* PhD thesis, Vanderbilt University, May 1991.

[NGS+01] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja, and N.-E. Bånkestad. Experiences from introducing state-of-the-artreal time techniques in the automotive industry. In *Proceedings 8th Int. Conf. and Workshop on the Engineering of Computer-Based Systems.* IEEE Computer Society Press, April 2001.

[OMG01a] OMG. Dynamic scheduling, joint final submission, Aug. 2001. orbos/01-04-01.

[OMG01b] OMG. Response to the OMG RFP for schedulability, performance, and time (revised submission), June 2001. OMG, RFP ad/2001-06-14.

[OMG01c] OMG. A uml profile for enterprise distributed object computing, June 18 2001. ptc/2001-12-04.

[OMG02] OMG. UML 1.4 with action semantics, 2002. OMG ptc/02-01-09.

[Pat00] T. Pattison. *Programming Distributed Applications with COM+ and Microsoft Visual Basic 6.0, 2nd edition.* Microsoft Press, July 2000.

[PEC] PECOS project. www.pecos-project.org/publications.html.

[RAJ01] E. Roman, S. Ambler, and T. Jewell. *Mastering Enterprise JavaBeans.* John Wiley and Sons, 2001. 2nd edition.

[Ray02] J. Rayfield. Keynote presentation. CASES/EMSOFT'2002, Oct. 2002.

[RNHL99] J. Reekie, S. Neuendorffer, C. Hylands, and E. Lee. Software practice in the Ptolemy project. Technical Report GSRC-TR-1999-01, Gigascale Silicon Research Center, April 1999.

[Sel02] B. Selic. Physical programming: Beyond mere logic. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proc. EMSOFT 2002*, volume 2491 of *Lecture Notes in Computer Science*, pages 399–406. Springer Verlag, Oct. 2002.

[SM91] W.H. Sanders and J.F. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE Journal on Selected Areas in Communications*, 9(1):25–36, 1991.

[St00] R. Soley and the OMG Staff Strategy Group. Overview of the proposed model driven architecture to augment the Object Management Architecture, 2000. omg/00-11-05, http://www.omg.org/docs/omg/00-11-05.pdf.

[StOG00] R. Soley and the O.S.S. Group. Model driven architecture (draft 3.2), Nov. 2000. OMG, White paper.

[StOG01] J. Siegel and the O.S.S. Group. Developing in OMG's model-driven architecture, Nov. 2001. OMG, White paper, Revision 2.6.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, N.Y., 1998.

[Tel] Telelogic Object Geode and TAU. http://www.telelogic.se.

[Ten00] David Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, 2000.

[USE01] *USE, a UML-based specification environment*, 2001. http://dustbin.informatik.uni-bremen.de/projects/USE/.

[vO02] R. van Ommering. Building product populations with software components. In *Proceedings of the 24th international conference on Software engineering*, pages 255–265. ACM Press, 2002.

[vOvdLK00] R. van Ommering, F. van der Linden, and J. Kramer. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[WBGP01] T. Weis, C. Becker, K. Geihs, and N. Plouzeau. A UML meta-model for contract aware components. In M. Gogolla and C. Kobryn, editors, *Proc. UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 442–456. Springer Verlag, Oct. 2001.

[WHSB98] Murray Woodside, Curtis Hrischuk, Bran Selic, and Stefan Bayarov. A wideband approach to integrating performance prediction into a software design environment. In *Proceedings of the first international workshop on Software and performance*, pages 31–41. ACM Press, 1998.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.

[WSO00] N. Wang, D. Schmidt, and C. O'Ryan. Overview of the CORBA Component Model, Sept. 2000. White paper.

[WZS02] Michael Winter, Christian Zeidler, and Christian Stich. The PECOS software process. In *Proc. Workshop on Components-based Software Development Processes, 7th Int. Conf. on Software Reuse*, April 2002.