MRTC

MÄLARDALEN REAL-TIME RESEARCH CENTRE

Classification and survey of component models

MRTC report ISSN 1404-3041
ISRN MDH-MRTC-242/2009-1-SE

MRTC PROGRESS project provides a part of funding for DICES project.

DICES (Distributed Component-based Embedded Software Systems) project has a goal to advance development of distributed embedded software systems, particularly with emphasis on software reusability and predictability of software quality. By adopting a component-based approach to engineering of embedded software systems, DICES aims to advance theories and methods for prediction of certain system properties (such as resource utilization, and performance), as well as to provide tools that will help in reusability of software components, and assure performance efficiency of systems.

PROGRESS and DICES project have many interests in common. Except for interest in the same systems domain (embedded systems), both projects are focusing on adaptation and extension of component-based development into a mature engineering discipline for efficient development of embedded software.

DICES technical report

Classification and survey of component models

Project name: DICES
Contract number: No. 03/07
Author(s): Juraj Feljan, Luka Lednicki, Josip Maras, Ana Petričić, Ivica Crnković

Executive summary

As component-based software engineering is growing and its usage expanding, more and more component models are developed. In this report we present a survey of software component models in which models are described and classified respecting the classification framework for component models proposed by Crnković et. al. [1]. This framework specifies several groups of important principles and characteristics of component models: lifecycle, constructs, specification and management of extra-functional properties, and application domain. This report analyzes a considerable amount of component models, including widely used industrial models, as well as research models.

Table of Contents

1 Introduction

Component models are a paramount concept in component-based software engineering, as they define how components are specified and connected. A large number of component models have been developed to date, but mostly having substantial differences in their approaches, meaning that a comparison between them by simply listing their properties is not achievable. Therefore, developing a systematic classification framework for component models is worthwhile.

In this technical report we use a classification framework proposed by Crnković et. al. [1] to classify a number of component models. In [1] these component models are given brief general overviews with their classification against the framework given only in tables. We expand this by providing a somewhat more detailed textual description of each model's classification, while also covering key properties of every model.

In Figure 1 we give a graphical representation of the classification framework. We also give tables of the classification. The figure and the tables are taken from [1] and are repeated here for the sake of completeness, and easier comprehension and readability. For details on the framework we refer the reader to the referenced article.

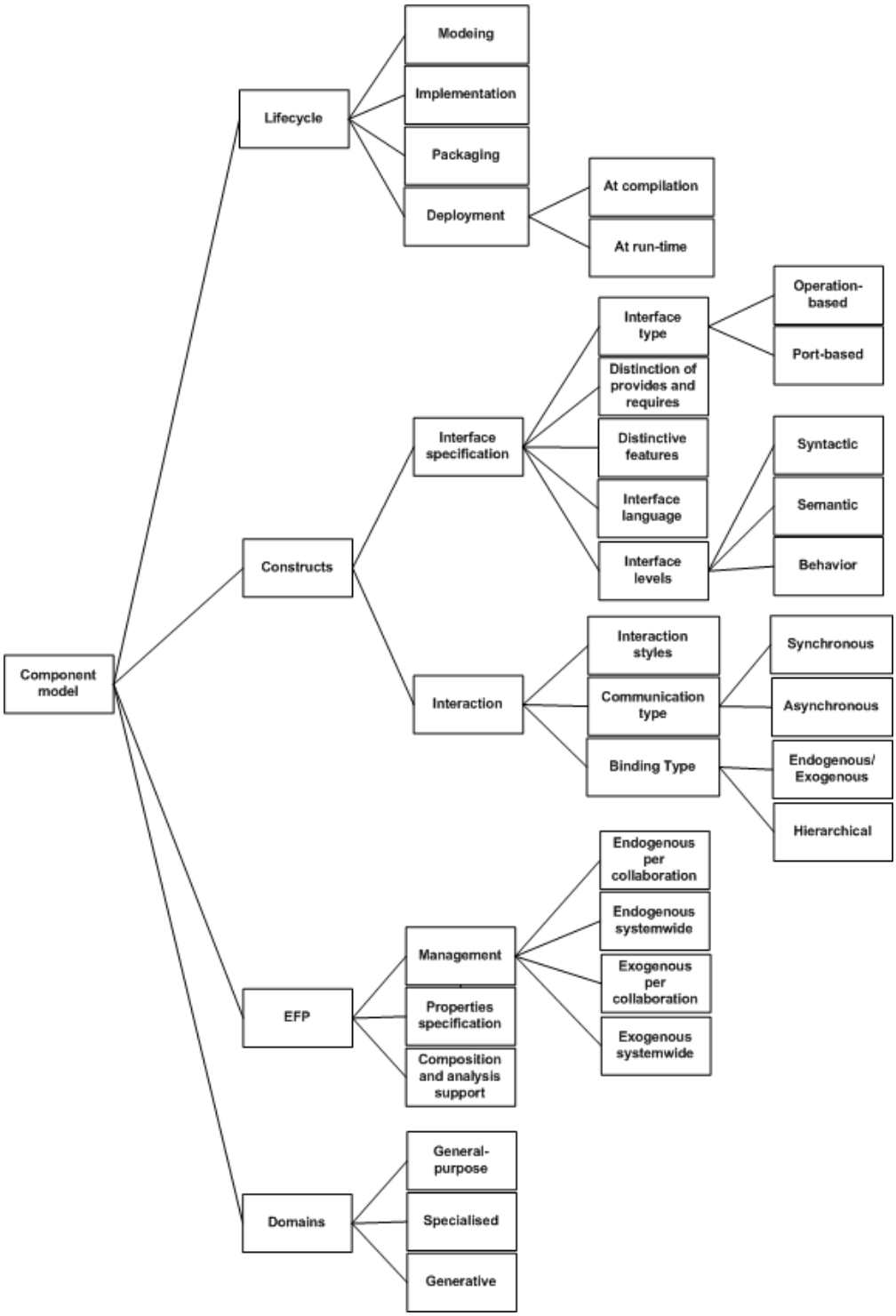

Figure 1: The classification framework visualized

Table 1: Lifecycle

Component models	Modeling	Implementation	Packaging	Deployment
AUTOSAR	use of virtual functional bus	C	non-formal specification of container	at compilation
BIP	a three-layered representation: behavior, interaction and priority	BIP language and C++	N/A	at compilation
BlueArX	ASCET-MD	C	packages	at compilation
CCM	N/A	language independent	deployment unit archive (DLLs, JARs)	at run-time
COM	N/A	language independent to some extent	DLL files, EXE files	at compilation, at run-time
COMDES II	ADL-like language	C	N/A	at compilation
CompoNETS	behavior modeling (Petri Nets)	language independent	deployment unit archive (DLLs, JARs)	at run-time
EJB	N/A	Java	EJB-Jar files	at run-time
Fractal	ADL-like language (Fractal ADL, Fractal IDL), Annotations (Fractlet)	Java (in Julia, Aokell) C/C++ (in Think) .Net lang. (in FracNet)	file system based repository	at run-time
IEC 61499	function block diagram	language independent	N/A	at compilation
JavaBeans	N/A	Java	JAR files	at compilation
Koala	CDL, IDL, DDL	C	component file based repository, interface file based repository	at compilation
KobrA	UML	language independent	file system based repository	at compilation
OpenCOM	N/A	language independent	N/A	at run-time
Palladio	domain-specific language for each role	Java	N/A	at run-time
PECOS	ADL-like language (CoCo)	C++ and Java	deployment unit archive (DLLs, JARs)	at compilation
Pin	ADL-like language	C	DLL	at compilation

	(CCL)			
ProCom	ADL-like language, timed automata	C	file system based repository	at compilation
ROBOCOP	ADL-like language, resource management model	C and C++	structures in zip files	at compilation and run-time
Rubus	Rubus Design Language	C	file system based repository	at compilation
SaveCCM	SaveCCM graphical language, XML adhering to the SaveCCM DTD, timed automata with tasks	C, Java	file system based repository, JAR files	at compilation
SOFA 2.0	meta-model based specification language	Java	repository	at run-time

Table 2: Constructs - interface specification

Component models	Interface type	Distinction of provides and requires	Distinctive features	Interface language	Interface levels
AUTOSAR	operation-based, port-based	yes	AUTOSAR Interface	C header files	syntactic
BIP	port-based	no	N/A	BIP language	syntactic, semantic, behavioral
BlueArX	port-based, operation-based	yes	Configuration Interface, Analytic Interface	XML adhering to the MSRSW DTD	syntactic
CCM	operation-based port-based	yes	facets and receptacles, event sinks and event sources	CORBA IDL, CIDL	syntactic
COM	operation-based	no	ability to extend interface	MIDL	syntactic
COMDES II	port-based	yes	N/A	C header files, state chart diagrams	syntactic, behavior
CompoNETS	operation-based, port-based	yes	facets and receptacles, event sinks and event sources	CoORBA IDL, CIDL, Petri nets	syntacic, behavior
EJB	operation-based	no	N/A	Java + annotations	syntactic
Fractal	operation-based	yes	Component Interface, Control Interface	IDL, Fractal ADL, or Java or C, Behavioural Protocol	syntactic, behaviour
IEC 61499	port-based	yes	event input and event output, data input and data output	XML	syntactic
JavaBeans	operation-based	yes	N/A	Java	syntactic
Koala	operation-based	yes	diversity interface, optional interface	IDL	syntactic
KobrA	operation-based	no	N/A	UML	syntactic, semantic, behavioral
OpenCOM	operation-based	yes	N/A	OMG IDL	syntactic

Palladio	operation-based	yes	inheritance, RDSEFFs, protocols	OMG IDL based	syntactic, semantic, behavior
PECOS	port-based	yes	ability to extend interface	CoCo language, Prolog query, Petri nets	syntactic, semantic, behavior
Pin	port-based	yes	N/A	Component Composition Language (CCL), UML statechart	syntactic, behavior
ProCom	port-based	yes	data- and trigger ports	XML based, timed automata	syntactic, behavioral
ROBOCOP	port-based	yes	ability to extend different types of interface/annotations	Robocop IDL (RIDL), protocol specification	syntactic, behavior
Rubus	port-based	yes	data- and trigger ports	C header files	syntactic
SaveCCM	port-based	yes	data-, trigger- and combined ports	XML adhering to the SaveCCM DTD	syntactic
SOFA 2.0	operation-based	yes	Utility Interface, possibility to annotate interface and to control evolution	Java, SPC algebra	syntactic, behaviour

Table 3: Constructs - interaction

Component models	Interaction styles	Communication type	Binding type	
			Exogenous	Hierarchical
AUTOSAR	request-response, message passing	synchronous, asynchronous	no	delegation
BIP	triggering rendezvous, broadcast	synchronous, asynchronous	no	delegation
BlueArX	sender-receiver, request-response	asynchronous, synchronous	no	delegation
CCM	request-response, triggering	synchronous, asynchronous	no	no
COM	request-response, events	synchronous, asynchronous	no	delegation, aggregation
COMDES II	pipe-and-filter	synchronous	no	no
CompoNETS	request-response	synchronous, asynchronous	no	no
EJB	request-response	synchronous, asynchronous	no	no
Fractal	multiple interaction styles	synchronous, asynchronous	yes	delegation, aggregation
IEC 61499	event-driven, pipe-and-filter	synchronous	no	delegation
JavaBeans	request-response, events	synchronous	no	no
Koala	request-response	synchronous	no	delegation, aggregation
KobrA	request-response	synchronous	no	delegation, aggregation
OpenCOM	request-response	synchronous	yes	no
Palladio	request-response	synchronous	yes	delegation
PECOS	pipe&filter	synchronous	no	delegation
Pin	request-response, message passing, triggering	synchronous, asynchronous	no	no
ProCom	pipe-and-filter, message passing	synchronous, asynchronous	yes	delegation
ROBOCOP	request-response	synchronous, asynchronous	no	no
Rubus	pipe-and-filter	synchronous	no	delegation
SaveCCM	pipe-and-filter	synchronous	no	delegation
SOFA 2.0	multiple interaction	synchronous,	yes	delegation

	styles	asynchronous		

Table 4: EFPs

Component models	Management of EFPs	Properties specification	Composition and analysis support
AUTOSAR	endogenous per collaboration (A)	N/A	N/A
BIP	endogenous systemwide (B)	timing properties	behavior composition
BlueArX	endogenous systemwide (B)	resource usage, timing properties	Reasoning Frameworks
CCM	exogenous systemwide (D)	N/A	N/A
COM	endogenous per collaboration (A)	N/A	N/A
COMDES II	endogenous systemwide (B)	timing properties	N/A
CompoNETS	endogenous per collaboration (A)	N/A	N/A
EJB	exogenous systemwide (D)	N/A	N/A
Fractal	exogenous per collaboration (C)	ability to add properties (by adding "property" controllers)	N/A
IEC 61499	endogenous per collaboration (A)	N/A	N/A
JavaBeans	endogenous per collaboration (A)	N/A	N/A
Koala	endogenous systemwide (B)	resource usage	compile-time checks of resources
KobrA	endogenous per collaboration (A)	N/A	N/A
OpenCOM	endogenous per collaboration (A)	N/A	N/A
Palladio	endogenous systemwide (B)	performance properties specification	performance properties at design-time
PECOS	endogenous systemwide (B)	timing properties, generic specification of other properties	N/A
Pin	exogenous systemwide (D)	analytic interface, timing properties	different EFP composition theories, example latency
ProCom	endogenous systemwide (B)	timing and resources	timing and resources at design- and compile-time
ROBOCOP	endogenous systemwide (B)	memory consumption, timing properties, reliability, ability to add other properties	memory consumption and timing properties at deployment
Rubus	endogenous systemwide (B)	timing properties	timing properties at design-time
SaveCCM	endogenous systemwide (B)	timing properties, generic	timing properties at design-

		specification of other properties	time
SOFA 2.0	endogenous systemwide (B)	behavioral (protocols)	composition at design

Table 5: Domains

Component models	Domain
AUTOSAR	specialized
BIP	specialized
BlueArX	specialized
CCM	general-purpose
COM	general-purpose
COMDES II	specialized
CompoNETS	general-purpose
EJB	general-purpose
Fractal	general-purpose, generative
IEC 61499	specialized
JavaBeans	general-purpose
Koala	specialized
KobrA	general-purpose
OpenCOM	general-purpose
Palladio	specialized
PECOS	Specialized
Pin	general-purpose
ProCom	specialized
ROBOCOP	specialized, generative
Rubus	specialized
SaveCCM	specialized
SOFA 2.0	general-purpose, generative

2 Overview of selected component models

2.1 AUTOSAR

AUTOSAR is a new standardized architecture created by a partnership of a number of automotive manufacturers and suppliers. The goal of AUTOSAR is to provide a way for managing increasing complexity of vehicular embedded systems, enable detection of errors in early design phases and improve flexibility, scalability, quality and reliability of such systems [2].

AUTOSAR defines a layered software architecture consisting of five layers. First three layers, Microcotroller Abstraction Layer, ECU Abstraction Layer and Service Layer sit on top of hardware and provide a standardized and hardware-independent interface to the AUTOSAR Runtime Environment. This Runtime Environment then supports the Application Layer, the AUTOSAR Component Model.

The main goal of AUTOSAR is to provide a standard for location independence and portability of software components for the automotive industry. Thus, the component model itself is not very advanced and does not fully reflect the capabilities of current state-of-the-art models [3].

During the development process, AUTOSAR provides some levels of system modeling by giving us the ability to interconnect components using a Virtual Functional Bus (VFB). The VFB provides an abstract level of viewing all communication mechanisms provided by AUTOSAR. In this way AUTOSAR enables early system integration that is independent of the physical allocation of components. At the time of deployment, the VFB is replaced by the AUTOSAR Runtime Environment that provides implementation for selected communication mechanisms.

During deployment of a system, AUTOSAR Software Components are compiled and linked into ECU specific executable. Although this provides a more efficient systems, ti also means losing the benefits of the component-based approach during run-time.

AUTOSAR Software Component package consist of implementation and component description. Implementation of a component can be either object code, or C source code. Component description consists of operations and data that the component provides and requires, requirements that the component has on the infrastructure, resources needed by the component and information about specific implementation of the component. Because of the hardware abstraction layer provided by AUTOSAR Runtime Environment the component's implementation is independent from the hardware infrastructure, e.g. type of microcontroller or ECU.

The AUTOSAR Software Components are defined as applications which run on the AUTOSAR infrastructure. These components are atomic, meaning that one component cannot be distributed over several AUTOSAR ECUs. An exception to this is *composition*, a logical interconnection of components packaged as a new component. The components inside the composition can be distributed over several ECUs.

A special type of AUTOSAR software components are sensor/actuator components. These components encapsulate dependencies on specific sensor or actuator hardware. They are dependent on a specific sensor or actuator, but independent of the ECU.

AUTOSAR Software Components interact with each other through their well-defined ports.

Services or data that a port provides or requires are defined by AUTOSAR Interfaces (which, accordingly, a port can provide or require). AUTOSAR Interfaces are described by C header files and cover only syntactical information.

Communication between components can follow either Client-Server (Request-Response) or Sender-Receiver (message passing) pattern. In case of Client-Server communication pattern providing port (server) implements operations defined by the interface, while the port that requires the interface (client) can invoke those operations. This type of communication can be either synchronous (if the client blocks its execution until the server returns a response) or asynchronous (in case the client does not block after the operation request is initiated). The Sender-Receiver pattern allows only asynchronous transfer of data. In this pattern the providing port (sender) generates the data and requiring port (receiver) has the ability to read this data. After the sender generates the data it doesn't wait or expect any response from the receiver. Type of communication is defined by the AUTOSAR interface that a port provides or requires.

Binding of AUTOSAR components is endogenous, having no separate connector entities. The connection between ports is managed by the ports themselves.

AUTOSAR allows use of compositions for sub-system abstraction. However, they are only used to group existing software components to manage complexity when designing logical system architecture [4]. They do not add any new functionality to that already defined by the components inside the composition, and do not have any binary footprint when deployed to ECU. Surface ports of a composite exposes can by explicitly defined by delegating ports of the aggregated components.

Although AUTOSAR Software Component descriptions have the ability to specify some extra functional properties, like resource (memory, CPU-time, etc.) that a software component requires, there is a lack of the capability to express the multitude of non-functional constraints, insufficient expressiveness of the interfaces [3]. In AUTOSAR, there is also a lack of ability to analyze properties of component composition, e.g. ability to guarantee that component's properties are preserved across integration, or that requirements of global properties of composed objects are meet.

2.2 BIP

BIP (Behavior, Interaction, Priority) **[5]** is a framework developed at Verimag used for modeling heterogeneous real-time components. In BIP the heterogeneity can refer to either: **synchronous** or **asynchronous** (or one of the variety of intermediate and hybrid models); and **timed** or **untimed** components.

Each BIP component is a superposition of three layers: **Behavior layer** which specifies a set of transitions; **Interaction layer** which uses a set of connectors that describe the interactions between the transitions of behavior and **Priority layer** which is used to define interaction priorities.

BIP components are defined with: a set of ports used for synchronization with other components which can also be used for data transfer; a set of control states defining internal component states; a set of variables used to store data and a set of transitions representing steps from one control state to the other while carrying an internal computation if a certain condition (so called guard) holds and synchronization happens on the specified port. Interfaces of BIP components are port-based and are defined on the syntactical, semantical and behavioral level.

For defining components BIP uses a mix of custom BIP syntax and C/C++ code.

```
atom::=
    component component_id
        port port_id+
        [data type_id data_id+]
        behavior
        {state state_id
            {on port_id [provided guard]
            [do statement] to state_id}+}+
        end
    end
```

Example 1: Defining atomic BIP component

Main BIP constructs are:

- atomic components whose behavior is specified with a set of transitions, but which have empty interaction and priority layers.
- connectors which are sets of ports of atomic components which can be involved in an interaction. Non-trivial interactions (interactions with at least two ports) denote a synchronization between included ports. BIP differentiates two basic synchronization modes:
  - Rendezvous or strong synchronization which includes all ports contained in a connector.
  - Broadcast or weak synchronization when all feasible interactions of a connector contain a particular port which initiates the broadcast.

```
interaction ::= port_id+
connector::=
    connector conn_id = port_id+
    [complete = interaction+]
    [behavior
        {on interaction [provided guard] [do statement]}+
    end]
```

Example 2: Connector definition

- Priority relations are used for prioritizing interactions.

Compound components are composed from either atomic components or other compound components by creating their instances with the addition of specifying connectors between them and priorities of their interactions. BIP is a hierarchical component model.

The process of system construction can be viewed as a series of formal transformations of the three

layers (Behavior, Interaction, Priority) where corresponding layers are separately composed.

BIP execution platform is partially implemented in the IF tool-set and the PROMETHEUS tool and also includes a front-end for parsing BIP and generating C++ code which can be executed and analyzed.

2.3 BlueArX

BlueArX[1] [6] is a domain-specific component model developed and used by Bosch for real-time embedded automotive applications, for example in engine control systems or chassis systems. These are closed control loop systems, meaning that they receive physical values from sensors, perform computations and then control actuators with new physical values.

BlueArX provides support in all stages of the lifecycle. Modeling is usually done using ASCET-MD[2] models. Implementation is done in C. Components are delivered as so called *packages*, and are both exchanged between Bosch teams and shipped to customers in this format. Each component consists of the specification, documentation and implementation. Deployment is done at compilation. BlueArX focuses on design-time (see *signal flows* below) and does not impose additional run-time overhead.

BlueArX supports two types of components: *atomic* and *structural*. An atomic component is a unit of specification that has an implementation (as stated earlier – in C), while a structural component is a unit of specification that has a decomposition (i.e. it is composed from several atomic and/or structural components). A structural component can export a subset of subcomponents' interfaces, in other words BlueArX provides support for hierarchical binding through delegation. The binding type is endogenous.

BlueArX divides interfaces into two types: *import* and *export*. An import interface specifies variables, messages, services calibration parameters, etc. required by a component to execute, while an export interface specifies the same types of elements that a component provides. Interface specification is done in XML adhering to the MSRSW (Manufacturer Supplier Relationship Software) DTD[3], and it includes computation methods, calibration parameters (maps, curves), physical units etc. The interface contractualization level is syntactic, but some semantic consistency checks are also preformed (e.g. unit consistency, value range). The interface type is both port-based (via messages) and operation-based (via service calls).

As distinctive features BlueArX has two additional types of interfaces: *Configuration* and *Analytic*. A Configuration Interface[4] specifies the variability of a software component, as it contains and describes all variant points and the dependencies between these variant points. For example, a component `C` supports either a turbo charger, or a compressor, or neither of them. This is expressed by the component having two variant points (`TURBO_CHARGER (boolean)` and `COMPRESSOR`

1 We thank Martin Herrmann from Bosch for providing additional information on BlueArX, and for his comments and review of this BlueArX overview.

2 A software modeling tool developed by the ETAS group, http://www.etas.com/en/products/ascet_md_modeling_design.php

3 Available at www.msr-wg.de/medoc/download/msrsw/v110/msrsw_v110-eadoc-en/msrsw-eadoc-en.pdf. MSR is a consortium of car manufacturers and suppliers that aims to enable process synchronization and proper information exchange based on XML

4 A concept still under development.

(`boolean`)), with three legal (`01`, `10`, `00`) and one illegal combination (`11`) between them.

An Analytic Interface is used to store components' EFPs. EFP values are specified in XML conforming to the MSRSW DTD. Since EPF values have dependencies to the hardware platform, compiler, software context etc., the context has to be specified. Analytic Interfaces in BlueArX systems are connected to Reasoning Frameworks, which are used for various EFP consistency checks. Thus, management of EFPs is endogenous systemwide.

Components communicate using messages (global variables) and service calls (function calls). Thus, the communication type is both asynchronous (when using messages) and synchronous (when using service calls). The interaction styles are sender-receiver (messages) and request-response (service calls).

BlueArX puts a special focus on the concept of *signal flows*. The idea is to use signal flow visualization to provide crucial behavior information on the component level, and get an explicit functional view from implicit component specifications. There are three types of signal flows:

- component internal flow,
- flow between components, and
- end-to-end flow (originates at a sensor, propagates through various components and ends at an actuator).

In many cases signal flows through a system depend on the mode of operation, meaning that the flow of information may change its path depending on the mode in which the system is operating. BlueArX provides support for *mode dependent signal flows*, which enables more precise flow analysis. Mode dependent signal flows manage the complexity of flow visualization by highlighting only those flows that are relevant for a particular mode.

2.4 COM

Component Object Model (COM) [7], [8] is a platform independent[5], distributed, object-oriented software architecture for creating and connecting binary software components, developed by Microsoft. It is a general-purpose component model, one of the most commonly used component models for desktop- and server side applications. Microsoft is applying COM to address specific areas such as controls, compound documents, automation, data transfer, storage, naming and so on [9].

The COM technology represents one of the earliest attempts (introduced in 1993) to increase program independence and allow programming language heterogeneity [10]. COM has its origins in OLE (Object Linking and Embedding), a technology that enables compound documents by maintaining active links between documents or even embedding one type of document within another. In 1996 COM was extended to support distribution, which resulted in DCOM (Distributed COM), Microsoft's answer to CORBA. In the same year, some parts of OLE were renamed to ActiveX. ActiveX components are used for creating distributed applications that work over the Internet through Microsoft's Internet Explorer browser. With Windows 2000, COM+ was released. COM+ added support for enterprise-level features (distributed transactions, resource pooling,

5 COM is mostly used in the Windows family of operating systems, however Unix implementations exist, and Microsoft provides an implementation for Macintosh systems.

disconnected applications, event publication and subscription, better thread management etc.) to COM. Today under the COM name the DCOM and COM+ technologies are included, and COM is used as the underlying architecture for OLE and ActiveX.

The COM platform has been superseded by .NET, and to some extent COM is now deprecated in favor of .NET, as Microsoft recommends that developers use .NET rather than COM for new development. However, COM is used in core parts of Windows and Microsoft Office, and continues to be supported as part of Windows.

COM provides support in the implementation, packaging and deployment stages of the lifecycle, while the modeling stage is not supported. COM is a binary standard, it applies after a component object[6] has been translated to binary machine code. Any language that produces binary compatible code, in a sense that the language can create structures of pointers and support method calling through pointers, can create and use component objects. These are for instance C, C++, Smalltalk, Ada, Pascal etc. This makes COM language independent to some extent. Component objects can be packaged either as EXE files or as dynamic-link libraries (DLL files). Deployment can be done either at compilation or at run time. The emphasis is on the former. This is enabled through the use of the `QueryInterface` method (details follow later).

One of the key principles of COM is that interfaces are specified separately from both the component objects that implement them and component objects that use them. An interface cannot be instantiated, a component object has to implement the interface and that component object is to be instantiated. Different component objects may implement the same interface differently. Thus, polymorphism fully applies to COM. A component object can, and typically does, implement more than one interface.

A client has access to a component object through a pointer to the object's interface. This interface pointer hides all aspects of the internal implementation of the component object, the object's data cannot be accessed, only interface methods can be called. Therefore, encapsulation also fully applies to COM.

Component objects are identified by class IDs (CLSIDs), while interfaces are identified by interface IDs (IIDs). Both CLSIDs and IIDs are globally unique, which eliminates any chance of collision that would occur with human-readable names and result in run-time failure. CLSIDs and IIDs are 128-bit integers.

COM interfaces are never versioned, which means that version conflicts between new and old component objects are avoided. A new version of an interface, created by adding more methods or changing existing ones, is an entirely new interface and is assigned a new IID. Therefore, a new interface does not conflict with an old interface even if all that changed is the semantics (but not even the syntax) of an existing method [9]. That way other component objects that rely on a particular interface can continue to work. New functionality is added to component objects by adding support for new interfaces. Since interfaces remain constant, their implementation can be altered without breaking component objects that use these interfaces.

Since component objects communicate through method calls, the interface type is operation-based. There if no distinction between the provides- and requires part of the interface. A distinctive feature of COM is the ability to extend interfaces. As the interface specification language a dialect of

6 COM's term for component, not to be confused with object-oriented source code objects, class instances.

Object Management Group's Interface Definition Language (OMG IDL) is used, the Microsoft Interface Definition Language (MIDL). The contractualization level of interfaces is syntactic.

The interaction style is request-response, this is determined by method calling. Also, DCOM and COM+ enable the event interaction style. The communication type is synchronous in the former case and asynchronous in the latter. Exogenous binding is not supported. There is support for hierarchical binding, through two techniques – delegation and aggregation. With delegation the composite object component "contains" the object subcomponent, and when the composite object component wishes to use the services of the object subcomponent, the composite object component simply delegates implementation to the object subcomponent, by delegating the method call to the object subcomponent's interface. In other words, the composite object component uses the object subcomponent's services to implement some of its own functionality (or possibly all of its own functionality). With aggregation the composite object component exposes interfaces of the object subcomponent as if they were implemented on the composite object component itself [9].

The binary standard enables COM to perform method calls transparently – all component objects, in-process, cross-process or remote, are available to clients in a uniform and transparent way.

All component objects implement the standard `IUnknown` interface, otherwise they are not component objects. All COM interfaces are derived from `IUnknown`. This interface defines three methods, `QueryInterface`, `AddRef` and `Release`.

`AddRef` and `Release` are used for reference counting. `AddRef` is called when a client is using an interface. `Release` is called when the client no longer requires use of the interface. When the reference count falls to zero, the component object can safely unload itself from the memory.

The `QueryInterface` method provides the mechanism for dynamic (run-time) discovery of capabilities of a specific component object. In other words, the method is used to find out whether or not an interface is supported by a component object. At the same time, `QueryInterface` is the mechanism that a client uses to get an interface pointer from a component object. When an application wants to use some function of a component object, it calls the object's `QueryInterface` method. If the component object supports the desired interface, it will return the appropriate interface pointer and a success code. Otherwise, it will return an error value. The application will then examine the return code – if successful, it will use the interface pointer to access the desired method [9]. The combination of immutable interfaces and `QueryInterface` allows development of applications in which component objects can be dynamically updated, without the need to update other component objects or recompile the application.

The Component Object Library is a part of the operating system which provides the mechanics of COM. It encapsulates the work associated with launching component objects and establishing connections between them. When an application wants to use a component object, it passes the CLSID of the component object to the Component Object Library. The Component Object Library uses that CLSID to look up the associated component object code in the registration database. The library then returns the object's class factory to the application. The class factory is used to instantiate the component object. Upon instantiation the class factory returns a pointer to the requested interface back to the calling application. The application neither knows nor cares where the component object is run, it simply uses the returned interface pointer to communicate with the object. The Component Object Library is implemented in the `COMPOBJ.DLL` file for newer

versions of Windows.

COM has no support for EFP specification or composition and analysis. Management of EFPs is endogenous per collaboration. This is rather understandable, as in COM's primary domains EFPs are not of great relevance.

To sum up, basic COM concepts include [9]:

- A binary standard for method calling between component objects.
- A provision for strongly-typed groupings of methods into interfaces.
- A base `IUnknown` interface providing:
  - A way for component objects to dynamically discover the interfaces implemented by other component objects (`QueryInterface` method).
  - Reference counting to allow component objects to track their own lifetime and delete themselves when appropriate (`AddRef` and `Release` methods).
- A mechanism to uniquely identify component objects and their interfaces (CLSIDs and IIDs).
- A "component loader" to set up component object interactions (Component Object Library).

2.5 COMDES-II

COMDES-II (COMponent-based design of software for Distributed Embedded Systems, version II) is a component-based software framework aimed for efficient development of reliable distributed embedded control systems with hard real-rime requirements [11], and gives solutions for this specific domain.

The methodology that COMDES-II defines provides the ability to model both architectural and behavioral aspects of systems. The goal of this modeling is to provide the ability to analyze and verify system behavior at high abstraction level, and enable automatic code generation which would reduce the effort of implementing the systems and minimize the errors introduced by manual coding. As a consequence of this code generation, components are deployed at compilation time.

COMDES-II defines a two-layer component model. Components in the first layer are called *actors*. Actors are active software artifacts consisting of multiple *I/O drivers*, which define their port-based interface, and a single *actor task*. I/O drivers can be either *communication drivers*, used for communication over network, or *physical drivers*, used for sensing or actuating from/to physical units. There is a distinction between input and output I/O drivers. Actors interact with each other by exchanging labeled messages. This interaction is asynchronous and follows a producer-consumer protocol known as content-oriented message addressing. There are no connecting entities between actors, leaving the binding endogenous.

In the second layer of component model, as specification of functional behavior of actor tasks, COMDES-II uses *function block instances*, which are instantiations of *function block types*. Function blocks are pure functional components implementing concrete computational or control algorithms, and communicate with each other with their port-like *inputs* and *outputs*. Function block types can be categorized as either *basic*, *modal, state machine* or *composite*. Basic function

blocks are elementary function blocks, from which more sophisticated kinds of function blocks can be constructed. Composite function enable for hierarchical composition of functional behavior. Their functionality is represented using function block diagrams consisting of interconnected function block instances. The connections between function blocks employ a synchronous data flow model of computation. State machine and modal function blocks are used together to specify sequential system behavior. State machine function blocks consist an event-driven state machine model, binary inputs that are used as events/guards for the state machine, and two outputs which signal the current state of the state machine and notify environment about changes of that state. To eliminate non-deterministic behavior transitions of the state machine are ordered using a number indicating the importance of a transition. State machines can also be historical. Output of state machine function blocks can be directed to one or more modal function blocks, changing the mode of operation of these modal function blocks and allowing for the change of functionality of the system depending on the state of a state machine function block. Modal function blocks are hierarchical in their nature; different modes of operation of a modal function are specified using function block diagrams.

At the actor level, systems demonstrate a *distributed timed multitasking* operation model. This allows for a system-wide specification and reasoning about timing extra-functional property. Verification of such properties using UPPAAL timed automata has been explored [12].

2.6 CompoNETS

CompoNETS is a simple general-purpose component model designed in an effort to combine component-based software engineering and high-level Petri nets, with the goal to provide a formal semantic framework for software components [13]. Benefits of this would be the ability to describe internal behavior of concurrent and distributed components, a formal, unambiguous semantics for features as event multicasting, and in the end having means to reason about compositions of components designed with this approach.

The component model itself is inspired by CORBA Component Model (CCM). However, it is more simple and precise, and focuses on behavioral semantics, thus more adequate for research. These simplifications allow for mapping between component model constructs and Petri net elements, thus providing the ability to model behavior of components and compositions of components.

Although the current implementation of CompoNETS is in Java, the model is not tied to any Java specifics, leaving the possibility of implementation in any programming language.

Interfaces of CompoNETS components are defined by *ports* that that either provide or require operations (*facets* and *receptacles*), act as an provide or require event data (*event sources* and *event sinks*), or provide access to configurable *properties*. These types of ports are chosen to cover interaction styles seen in many component-based technologies used in industry: multicast asynchronous event-based communication, synchronous method invocation, and design-time configuration. Ports are defined using Java interfaces.

CompoNETS allows for hierarchical structure by providing ability to view assemblies of components as new components by hiding, promoting or renaming lower lever features at the upper level.

There is no support for extra-functional properties in CompoNETS.

2.7 Corba Component Model (CCM)

CORBA Component Model is a part of the CORBA 3 standard defined by Object Management Group (OMG). Its purpose is to reduce the effort of developing and deploying CORBA applications [14]. By using CORBA middleware as a base for component model, the CCM technology inherently provides a good method for defining software components and connecting such components into distributed systems. The CCM standard is focused on implementation of applications and doesn't provide any modeling capabilities.

Components in the CCM can be implemented in any programming language and on any platform as long as they use the CORBA middleware. A component package in the CCM consists of compiled program code (e.g. library file for C++ or class file for JAVA), and a CORBA *component descriptor*. Component descriptor is an XML file that contains information about the interfaces and services that the component supports.

The CCM components are deployed at run-time into CCM framework elements called *containers*. The deployment is done using *assembly archives*, which contain component archives, component property files and a component assembly descriptor. During run-time the container provides a framework for components that hides any specifics of the underlying platform. services like component life-cycle management, naming, transactions and security to the components it hosts, relieving component developers and application builders of the burden of implementing these functionalities. Operations for life-cycle management of components are isolated from the components themselves and implemented by objects called *component homes*. Each component type has to have a component home that is associated with the type. CCM development process also supports automated code generation using CCM *Component Implementation Framework* (CIF). For this purpose CCM also defines a declarative language called *Component Implementation Definition Language* (CIDL) for describing components. A CIF compiler reads component descriptions made using CIDL and generates some parts of component implementation (e.g. introspection, activation, state management) and CORBA component descriptors used in packaged components.

CCM defines two levels of components: *basic* level and *extended* level. Basic components have operation-based interfaces and provide a way to wrap regular CORBA objects. Components on this level are also fully compatible with Enterprise JavaBeans (EJB) component model specification, allowing for easier integration and mapping between these two models [15]. Extended components provide a much richer set of functionality. Their interfaces are defined by *ports*. CCM extended components support five different types of ports:

- **Facets**, which declare interface implemented by components,
- **Receptacles**, which explicitly represent which interfaces a component uses,
- **Event sources**, which enable components to be emitters or publishers of events,
- **Event sinks**, through which components declare that they accept events from event sources,
- **Attributes**, exposed named values that are primarily intended for component configuration.

Using these types of ports, component implementer can define that a component provides or requires data or services using synchronous or asynchronous communication. Interfaces of both basic and extended components are defined using *Interface Definition Language* (IDL). IDL definitions cover only syntactical information. Connection between ports are handled endogenously,

by components themselves. CCM does not support hierarchical composition.

CCM defines four common patterns for implementing components called *component categories*. These categories are [14]:

- **Session**. Session components are temporary objects that do some work on the behalf of a client. They are often created for each client that connects to a server and discarded when the client disconnects. Session components are stateful.
- **Service**. Service components are stateless temporary objects that provide services to clients. One service component can service multiple clients.
- **Entity**. Entity components have a persistent state managed by the component container. They are mainly used to represent data stored in a database.
- **Process**. Process components have, like entity components, persistent state. They are used for representing business processes where the state of the process needs to be stored persistently or the process has to participate in a distributed transaction.

CCM does not provide any support for extra functional properties.

2.8 EJB

Enterprise JavaBeans (EJB) is a component model developed by Sun Microsystems with current version 3.0 [16]. EJB has quite limited scope but despite its limitations, it has been widely used and popular in Java community. EJB is primarily used for a client – server model of distributed computing. It envisions the construction of object-oriented and distributed business applications. The model simplifies the development of middleware by providing server support for a set of services, such as transactions, security, persistence, concurrency and interoperability.

The EJB component model logically extends the JavaBeans [17] component model to support *server components*. Server components are reusable, prepackaged pieces of application functionality that are designed to run in an application server. They are similar to development components, but they are generally larger grained and more complete than development components. EJB components (*enterprise beans*) cannot be manipulated by a visual Java IDE in the same way that JavaBeans components can. Instead, they can be assembled and customized at deployment time using tools provided by an EJB-compliant Java application server.

2.8.1 Constructs

EJB components

An enterprise bean is a reusable, portable J2EE component which consists of methods that encapsulate business logic, and run inside an EJB Container. EJB components are limited to Java programming language, but they may be invoked from various other languages e.g. C++, C#, Visual Basic .NET. The EJB 3.0 bean class can be a pure Java class often referred as POJO and the interface can be a simple business interface.

EJB specification introduces three kinds of components called *beans*: *Entity beans*, *Session beans* and *Message – driven beans*.

Entity beans

An entity bean is a complex business entity which represents a business object that exists in the database. Its purpose is to access to data remotely over network. Each entity bean represents an object view on one record from the database and is defined by primary key. Entity beans may be shared between multiple users that use a primary key to access a particular bean. Invocations are performed synchronously. Entity beans are state full due to permanent storage background.

Entity beans introduced in EJB 3.0 specification are represented by Java Persistence API [18] entities, and they differentiate from the concept of entity beans that existed in previous EJB specifications (EJB 1.x, EJB 2.x). The EJB 1.x and 2.x entity beans must conform to a strict component model. Each bean class must implement a home and a business interface. The EJB 1.x and 2.x container requires very detail XML configuration files to map the entity beans to tables in the relational database. All these requirements are the reason why entity beans were obviated by software developers.. Introducing of entity beans as POJOs, made EJB 3.0 much more eligible an it simplified enterprise Java development with EJB.

Session beans

Session beans perform a task for a client; optionally they may implement a web service. Contrary to entity beans, session beans are not permanent and have no primary key since they are not backed by a database or other form of permanent storage.

Session beans are not shareable, as each session bean represents a single client inside the application server. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server. Invocations of session beans are synchronous.

Session beans may be statefull or stateless. Statefull bean maintains its state across different method calls through its instance variables which represent the state of a unique client-bean session. As a consequence, statefull session bean can be used by one remote client at a time. Stateless bean does not hold its state, when a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client, but only for the duration of the invocation. Except during method invocation, all instances of a stateless bean are equivalent, therefore stateless beans may be used by more than one remote client at a time.

Message – driven beans

Message-driven beans act as a listener for a particular messaging type, such as the Java Message Service (JMS) API. Similar to session beans, message-driven beans do not represent any data directly, however they may access any data in an underlying database. The most visible difference between message-driven beans and session beans is that clients do not access message-driven beans through interfaces. In other words, client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through some messaging service (for example JMS). Message-driven beans are executed when a message from a client arrives on a server, this means that their invocation in asynchronous. A single message-driven bean can process messages from multiple clients.

EJB Interfaces

An interface of an enterprise bean is specified as a set of methods and attributes, using Java programming language. Unlike session beans, message-driven and entity beans do not have interfaces that define client access because they have a different programming model.

A client can access a session bean only through the methods defined in the bean's business interface. All other aspects of the bean (method implementations and deployment settings) are hidden from the client. Session beans can expose one of two kinds of interfaces:

- remote interface: represents provisions of a bean. Provides an access point for a remote client and defines the business and life cycle methods that are specific to the bean
- local interface: defines the bean's business and life cycle methods that allow only local access (a local client must run in the same Java virtual machine (JVM) as the enterprise bean it accesses)

Each session bean has to implement at least one interface (remote or local). Although it is uncommon, it is possible for an enterprise bean to allow both remote and local access. Both kinds of bean interfaces are provided interfaces. EJB does not support required interfaces of a bean.

Message-driven beans and entity beans can also define and implement some interface, but it is not obligatory.

In addition, bean class can, but is not required to implement interfaces that it defines. However, implicitly, the interface of an enterprise bean is a set of the methods it implements and its attributes.

In order to additionally specify an enterprise bean, EJB 3.0 uses metadata annotations which are inspected by service framework. The EJB 3.0 specification itself defines a wide range of annotations that cover different attributes such as transaction or security settings, object-relational mapping and injection of environment or resource references. Metadata annotations are also used to specify the bean or interface and run time properties of enterprise beans. For example, a Session bean is marked with @Stateless or @Stateful to specify the bean type, message-driven beans are marked with @MessageDriven annotation.

As an alternative to Java annotations, there are deployment descriptors which were also used in previous versions of EJB (EJB 1.x, EJB 2.x). Deployment descriptor is an XML file which can be used to override some annotations, but also for describing application level metadata.

Composition of constructs

It is important to mention that EJB does not support connection-oriented programming, but follows traditional object-oriented composition (third party can not bind EJBs, but an EJB can specify dependencies to other components). Binding of enterprise beans is performed at runtime. In addition the composition specification of EJB components is location-transparent; the run-time location of components (placed on a local or a remote node) is specified separately from the binding information. A strength of EJB is automatic composition of component-instances with appropriate services and resources that component-instances are dependent on. This includes automatic configuration of necessary implicit middleware services based on needs specified by annotations or in the deployment-descriptor (transactions, persistence and security)

Communication between beans or between client and a bean is performed using Remote Method Invocation [19], which is a Java implementation of a Remote Procedure Call. Communication between enterprise beans is managed by JVM.

An example of an EJB component system and its interaction with clients is shown on Figure 1..

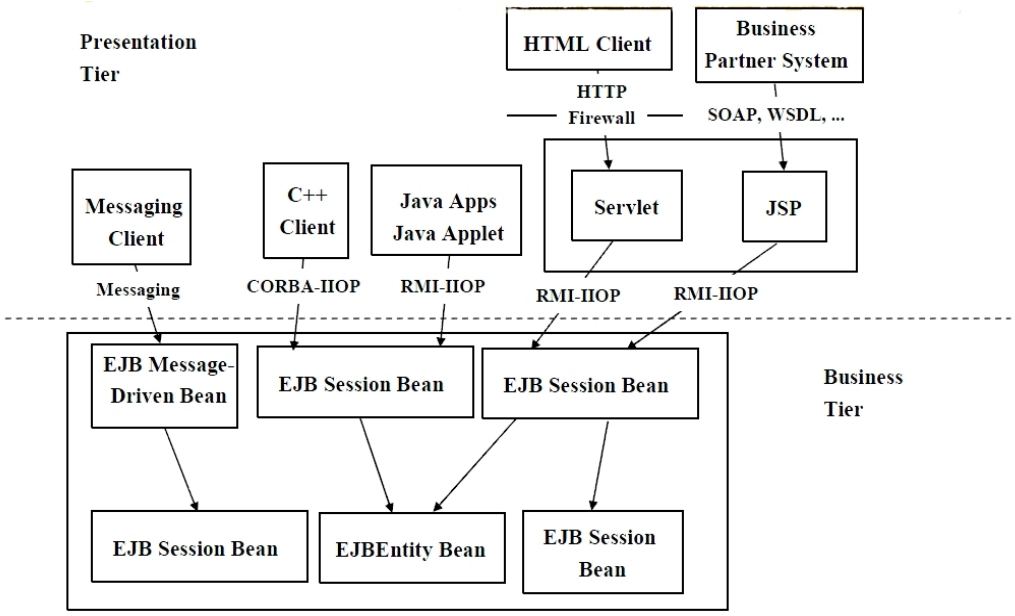

Figure 2: Client interaction with EJB component system

2.8.2 Life cycle

Packaging

EJB are packaged into an EJB JAR file, the module that stores the enterprise bean. An EJB JAR file is portable and can be used for different applications. To assemble a Java EE application, one or more modules (such as EJB JAR files) are packaged into an EAR file, the archive file that holds the application.

Deployment

EJB beans are deployed in an EJB Container which is in charge of their management at runtime (start, stop, passivation or activation) and extra-functional properties (such as security, reliability, performance). The Container can hide to application programmers some of the complexities inherent in the handling of non-functional aspects in a software system, such as distribution and fault-tolerance.

2.8.3 Extra-functional properties

EJB is primarily aiming at industrial use and it has been designed to support component developers at an implementation level, while lacking the sufficient support for specifying or analyzing extra-functional properties.

2.8.4 Benefits of Enterprise Beans

For several reasons, enterprise beans simplify the development of large, distributed applications. First, the EJB Container provides system-level services to enterprise beans so the bean developer

can concentrate on solving business logic problems. The EJB container, rather than the bean developer, is responsible for system-level services such as transaction management and security authorization.

Another benefit is that enterprise beans contain the application's business logic, therefore the developer of an enterprise bean client can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Due to the fact that enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant Java EE server provided that they use the standard APIs.

2.9 Fractal

Fractal [20] is a component model developed by France Telecom R&D and INRIA. The main goal of Fractal is to provide an extensible, open and general component model that can be tailored to fit a large variety of applications and domains. It tends to cover the whole development life cycle from design and implementation, up to deployment and maintenance/management (i.e. monitoring and dynamically reconfiguring) of various complex software systems.

Fractal can be used with any programming language and can be applied to variety of systems and applications from operating systems, middleware platforms to graphical user interfaces. Fractal currently provides different instantiations and implementations such as a C-implementation called Think [21], which targets especially the embedded systems and a reference implementation called Julia [22] written in Java.

Fractal component model includes several important features, such as:

- nesting components into *composite components*
- reflectivity – component has introspection capabilities, it can expose its externals and internals to other components, and it may respectively be created from other components
- component sharing – a given component can be included (shared) by more than one component
- binding components – component connections are represented by a single abstraction called *bindings* which covers any communication style such as synchronous method calls or remote procedure calls
- execution model independence
- extra-functional properties associated to a component can be customized

2.9.1 Constructs

Fractal components are represented through operation-based interfaces they expose. Every component, besides its functionality, can include a set of control capabilities. These capabilities are not fixed in the component model, but can be extended and adapted to fit programmer's constraints and objectives. Therefore the Fractal component model is defined as an extensible system of relations between well defined concepts and corresponding APIs that Fractal components *may* or *may not* implement, depending on what they can or want to offer to other components. This set of

specifications can be organized as increasing "levels of *control*":

- At the lowest level, a Fractal component does not provide any control capability and can be used only by invoking operations provided by the component.
- At the next level which is called the external "introspection" level, a Fractal component can provide introspection functions to introspect its *external* features (in other words, to explore operations and control capabilities that the component provides or requires).
- At the top level of control capability, which can be called the "configuration" level a Fractal component can provide ways to introspect and reconfigure its content (to manage the set of its subcomponents, bindings between these components etc.).

It is important to note that in Fractal everything is optional (a component can, but is not required to provide specific capabilities), also everything in Fractal is extensible and can be customized to fit specific needs. In Fractal specification, frameworks and features are specified using the interface declarations (called *language interfaces*) that a component should implement in order to reach certain control capabilities. Within this overview, some of these interfaces and their purpose will be mentioned, for further details see the Fractal specification [23].

The advantage of extreme modularity and extensibility of the Fractal model is that it can be applied to many situations. The drawback is that two arbitrary Fractal components will generally not be able to work together, because they may use very different, and potentially incompatible, options or extensions of the Fractal model.

Interfaces

Fractal defines a *component interface* as "an access point to a component, i.e., a place where operation invocations can be emitted or received". The component interface implements a *language interface* which is made of several operation declarations and it represents a type. Fractal component interfaces can be defined either directly in *any* programming language (e.g. Java, C), or indirectly via *any* IDL. As a consequence, the constraints and costs associated to the use of an IDL do not have to be paid for, if interoperability between distinct Fractal components is not needed.

When seen as black box, i.e. when its internal organization is not visible, the only visible details of a Fractal component are access points to this black box, called its *external interfaces* (see Figure 3).

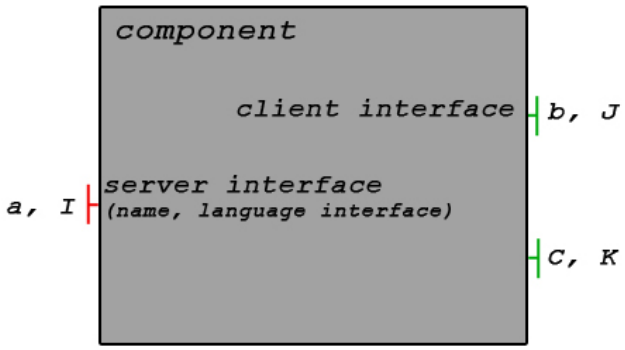

Figure 3: External view of a Fractal component

One may distinguish two kinds of interfaces: a *client* (or *required*) interface emits operation invocations, while a *server* (or *provided*) interface receives them. In addition, Fractal distinguishes between a functional and a control interface. A *functional* interface is an interface that corresponds

to a provided or required functionality of a component, while a *control* interface is a server interface that corresponds to a "non functional aspect", such as introspection, configuration or reconfiguration, and so on.

Each interface can have a name, in order to distinguish it from the other interfaces of the component (a component can have several interfaces implementing the same language interface) . To do so, Fractal offers a framework based on names and naming context. A name identifies a component interface and is always associated to a naming context. The name is generally invalid outside the context, for example the naming context of a Java reference is the Java Virtual Machine (JVM) in which the designated object resides, this name is meaningless outside this context and, in particular, in another JVM. A name and its context are represented by `Name` and `NamingContext` interfaces which should be implemented by the component in order to use the framework. These interfaces allow to manage the name of a component (e.g. serialize it) and to create new ones.

Internal component structure

Internally, a Fractal component is formed out of two parts: a *controller* (also called membrane), and a *content* (see Figure 4). The content of a component is composed of (a finite number of) other components, called *sub components*, which are under the control of the controller of the enclosing component. The Fractal model is thus recursive and allows components to be nested (i.e. to appear in the content of enclosing components) at an arbitrary level.

A component that exposes its content is called a *composite* component (configuration level). A component that does not expose its content, but has at least one control interface, is called a *primitive* component (external introspection level). A component without any control interface is called a base component (first level of control).

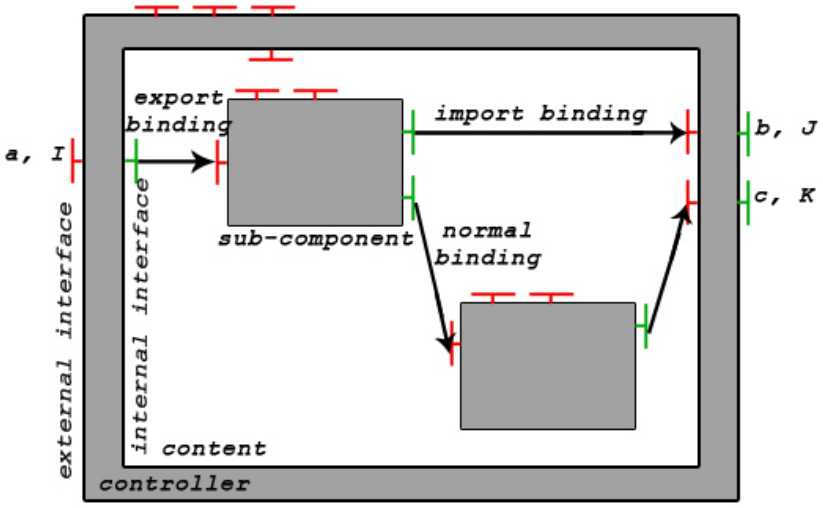

Figure 4: Internal view of a Fractal component

The controller of a component can have *external* and *internal* interfaces. External interfaces are accessible from outside the component, while internal interfaces are accessible only from the component's sub components. The controller embodies the control behaviour and in particular it can:

- Provide an explicit and causally connected representation of the component's sub

components.

- Intercept oncoming and outgoing operation invocations targeting or originating from the component's sub components.
- Superpose a control behavior to the behavior of the component's sub components, including suspending, check pointing and resuming activities of these sub components.

A Fractal component may appear in the content of (be *shared* by) several distinct enclosing components. A component that is shared among two or more components is under the control of their respective controllers. The resulting configuration (e.g. which control behavior is enacted) is in general determined by an encompassing component that encloses all the relevant components in the configuration.

Composition of constructs

A *binding* in Fractal, is a communication path between component interfaces, whence to access the interface designated by a name, a binding must be established to this interface. For example, in order to access a remote interface designated by an CORBA Interoperable Object Reference (CORBA IOR), a socket must be opened to send an invocation message to the remote interface. Bindings are created by *binders*. A binder is represented by the `Binder` interface, which allows to create a binding to the interface designated by the given name. By specifying the `Binder` interface, Fractal supports all kinds of communication and interaction styles.

The Fractal model distinguishes between *primitive bindings* and *composite bindings*. A primitive binding is a binding between one client interface and one server interface, in the same address space, which means that the operation invocations emitted by the client interface should be accepted by the specified server interface.

A composite binding is a communication path between an arbitrary number of component interfaces, of arbitrary language types. These bindings are represented as a set of primitive bindings and *binding components* (stubs, skeletons, adapters etc..). A binding component is a normal Fractal component, whose role is dedicated to communication. Binding components are also called *connectors*.

Component introspection and control

The interfaces of a component can be introspected (external introspection level) with two language interfaces, `Component` and `Interface`: one to get the list of interfaces of a component, and one to introspect the interfaces themselves. These two interfaces are of course optional, as everything in the Fractal model.

At the configuration level, a component can offer various controlling features. A component can implement interfaces such as:

- the `ContentController` interface to add and remove sub components of this component
- the `BindingController` interface to bind and unbind its client interfaces to other components through *primitive* bindings
- the `LifeCycleController` interface to help and support changing of a component (changing an attribute or a binding, or removing a sub component) while it is executing, since dynamic reconfigurations can cause the inconsistent application state or lost of data.

Instantiation

Except for the frameworks to introspect, configure and reconfigure existing component, Fractal defines a framework to create new components. In the instantiation framework, components are created by other components called component *factories*. The Fractal model distinguishes between generic component factories, which can create several kinds of components, and standard component factories, which can create only one kind of components, all with the same component type. In addition Fractal has a special kind of standard factory component that creates components based on a *template*.

2.9.2 Extra-functional properties

Extra-functional properties of Fractal components are supported through the notion of an attribute. As defined in Fractal specification; an *attribute* is a configurable property of a component. Every attribute can be read and changed, in order to read and write its attributes from outside the component, a component can provide an `AttributeController` interface. Having attribute compositions and analysis is not considered in the Fractal specification, but due to the extensibility of Fractal model, it can be supported.

2.10 IEC 61499

IEC 61499 standard has been developed by the International Electrotechnical Commission (IEC) to support the development of automation and control systems. It has evolved from the IEC 61131-3 standard that is widely used in development of software for Programmable Logic Controllers (PLCs).

An IEC 61499 *systems* consist of *devices,* which in turn consist of *resources* and interfaces to a communication network [24]. A resource is an element that independently executes a part of an IEC 61499 *application*. One application can be distributed, meaning that they can be deployed over several resources or devices. IEC 61499 supports component-based approach only during design time as applications are deployed as compiled executables. Implementations of IEC include languages like Java [25], C++ [26], or other.

IEC 61499 applications are built from reusable software components called *Function Blocks* (FBs) by connecting these FBs into *Function Block Networks*. FBs encapsulate a part of applications functionality and expose is through their explicit interfaces, leaving no *hidden* interface.

Interfaces are defined by port-like inputs and outputs. Inputs and output can either handle control flow (events) or data flow. Definition of inputs and outputs is on the syntactical level. Execution of the FBs is event driven, while the data is transferred using the pipe-and-filter pattern. All interaction between FBs is synchronous. Connections between inputs and outputs is handled by the components themselves.

There are three types of FBs defined by IEC 61499:

- **Basic function blocks (BFB)**. The behavior of a BFB is defined by an event driven state machine called *Execution Control Chart* (ECC). Whenever the active state of ECC changes, an action that is associated with the new state is executed. This action can either be an algorithm written in any programming language, an output event, or both.

- **Composite Function Blocks (CFB)**. The behavior of a CFB is defined by a FB network that consists of instances of any FB type, parameters and connections between FBs. In this way a developer can accomplish hierarchical composition of FBs.
- **Service Interface Function Blocks (SIFB)**. SIFBs are a way for encapsulating the interaction with external elements not defined by IEC 61499. They enable developers to wrap this external functionality and provide same interface as all other FB types. Although their functionality is hidden, it can by described using sequence diagrams.

Definitions of FBs, including their interfaces and behavior definitions, are stored in XML files.

IEC 61499 standard does not provide any facilities to specify or reason about extra functional properties.

2.11 JavaBeans

The *JavaBeans technology*[7] [17] is a portable, platform-independent software component model for the Java Standard Edition platform. The technology was introduced in 1997 and is developed by Sun Microsystems. The current version of the specification is 1.01 from year 2002, which includes some minor changes to the original document. The technology consists of a Java package (`java.beans`) and the JavaBeans specification which describes how classes and interfaces from the package should be used to implement the Java bean[8] concept. In simple words, a Java bean is a Java class that complies with conditions stated in the JavaBeans specification.

JavaBeans is a general-purpose component model, widely accepted and used, mostly in the desktop- and Web application domains. It focuses on making small lightweight components easy to implement and use, while making heavyweight components possible. Basic JavaBeans concepts can be learned very quickly and little effort is needed to start writing and using simple beans.

The specification defines Java beans as reusable software components that can be manipulated visually in a builder tool. The visual manipulation is one of the strongest aspects of the technology, as it allows "visual programming" by dragging beans from a palette onto a workspace where they can be configured and connected to other beans, thus enabling easy and intuitive development of applications. This is especially suitable for building graphical user interfaces (GUIs), where beans are most commonly employed. For instance, all GUI components, such as buttons, panels, check boxes etc., from Swing, the Java SE GUI framework, are beans. Currently, two most used bean compliant tools are Eclipse [27] and NetBeans [28]. However, although beans are primarily targeted at builder tools, they are also entirely usable by human programmers, as their use is not dependent on tools.

Each Java bean has to be able to run in two different environments. First, a bean needs to be capable of running inside a builder tool, it must be able to provide the builder tool with design information, so a user is able to configure it. This is referred to as the *design environment* or *design-time*. For this configuration process a lot of extra baggage (metadata, property editors, customizers, icons

7 It is important to differentiate between JavaBeans and Enterprise JavaBeans. Both technologies are software component models, use a similar name and are implemented in Java. However, their purpose and architecture are different.

8 The term "JavaBeans" stands for the technology, while the term "Java bean" or simply "bean" signifies a particular software component that conforms to the JavaBeans component model.

etc.) is carried by the bean. In addition, a bean must be able to be used during *run-time* within a generated application. During run-time there is much less need for customization of the behavior and appearance, so a bean carries less baggage than during design-time.

Many beans have a strong visual aspect, but while this is common, it is not required. Beans can be *visual* or *non-visual* (*invisible*). The GUI representation of beans may be the most obvious and compelling part of the JavaBeans technology. However, it is possible to implement non-visual beans that have no GUI representation. These beans are still able to call methods, fire events etc. They are also represented visually in a builder tool, so they can be configured. They simply have no screen appearance of their own. In other words, non-visual beans are invisible only at run-time, but are visible during design-time. Visual beans are visible both during design-time and run-time.

JavaBeans provides support in the implementation, packaging and deployment stages of the lifecycle, while the modeling stage is not supported. The implementation language is, as expected, Java. Beans are packaged in JAR (Java Archive) files. These are archive files in ZIP format. One JAR can contain one or more beans. A JAR containing beans must have a manifest file, which describes the beans in the JAR. Each JAR holding beans includes the following:

- Class files representing beans. These entries must have names ending in "`.class`".
- Optional source code files of beans. These entires have names ending in "`.java`".
- Optional serialized prototypes of beans. These entries must have names ending in "`.ser`".
- Optional help files in HTML format to provide documentation for the beans.
- Optional internationalization information to be used by beans to localize themselves.
- Other resource files needed by the beans (images, sound, video etc.).

Deployment, i.e. integration of beans into systems is done at compilation time.

Individual Java beans vary in functionality, but have the following typical common features:

- properties,
- events,
- methods,
- customization,
- introspection, and
- persistence.

A *bean property* is a named attribute of a bean that can affect its behavior or appearance. Examples of bean properties include color, label, font etc. Properties can have arbitrary types, including both primitive types and class or interfaces types. Properties are accessed via method calls on the owning bean.

There are four types of properties defined in the JavaBeans specification:

1. simple,
2. indexed,

3. bound and
4. constrained.

A *simple property* has a single value whose changes are independent from other properties.

Indexed properties support a range of values instead of a single value. It is possible to read or write a single element or the whole array corresponding to the indexed property.

Sometimes when a bean's property changes, another object might need to be notified of the change and react to it. These are *bound properties*. Whenever a bound property changes, a notification of the change is sent to interested listeners. Bound properties are normally used when a number of beans want to keep a shared value. For instance, for maintaining a common background color. When one bean changes its background color, the change is then promoted to all beans registered as listeners of that property change. That way they can adjust their background colors too.

Constrained properties are similar to bound properties. When a constrained property changes, an event is generated. However, the change is not necessarily accepted by the listeners, it first needs to be validated. If the change is not appropriate for a listener, it can be rejected, keeping the old value of the property.

Beans use the Java Event Model for communication. *Events* provide a convenient mechanism for allowing beans to be plugged together in a builder tool. For a bean to be the source of an event, it must implement methods that add and remove listeners for a particular type of event. For a bean to receive an event, it must implement an event listener interface[9].

The *methods* of a bean are normal Java methods which can be called from other objects. A bean's methods represent its interface, through which the bean can be accessed and manipulated. Since the interface of a bean consists from the bean's methods, the interface type is operation-based. There is a distinction between the provides- and requires part of the interface. The provides part consists of the bean's methods., i.e. the bean offers its methods to be called from other beans. In order for a bean to be the listener of another bean's property changes, the listener has to implement a listener interface, i.e. implement a particular method. That way the source bean defines which methods it requires from the listener beans.

There are no distinctive features which bean interfaces introduce and which don't exist in other component models. The interface language is inherently Java, and the contractualization level is syntactic. Through methods JavaBeans supports the request-response interaction style. Additionally, thanks to the event communication model, JavaBeans supports the event interaction style. The communication type between beans is synchronous. JavaBeans does not support exogenous nor hierarchical binding.

When a user is composing an application in a builder tool, he needs to be able to configure the beans he is using. *Customization* is the process of modifying the appearance and behavior of a bean within a builder tool, so that the bean meets the user's specific needs. Customization is done at design-time.

Introspection is the automatic process of analyzing a bean to reveal its properties, events and

9 This is a Java interface. It is not to be confused with the interface of the bean, which consists from the methods of the bean. These two terms (Java interface and bean interface) overlap but are not entirely the same. When a bean implements a Java interface by implementing methods proscribed in the Java interface, these methods become part of the bean interface.

methods. Introspection is used by builder tools. By uncovering beans' properties, events and methods, tools provide support for easy visual manipulation of beans. The simplest way to enable introspection is to write beans' source code by following particular design patterns, i.e. by using conventional names and type signatures for methods.

Persistence refers to the characteristic of data to outlive the execution of the program that created it. The mechanism that makes persistence possible is called *serialization*. Object serialization means converting an object into a data stream and writing it to storage. A serialized object can then be reconstructed by *deserialization*. All beans are required to support serialization.

JavaBeans has no support for EFP specification or composition and analysis. Management of EFPs is endogenous per collaboration. Regarding the envisioned domain of JavaBeans (desktop- and Web applications) the lack of EFP support is understandable.

2.12 Koala

Koala is a specialized component model and architectural description language developed by Philips. It targets embedded software, more specifically consumer electronics. Philips software architects and developers use it to develop software for their mid- and high-range TV sets.

Koala is intended to handle the diversity and complexity of embedded software, at an increasing production speed, by using and reusing software components within an explicit software architecture [29]. It does not address third-party development or run-time reconfiguration mechanisms (like for instance OpenCOM), it rather focuses on consumer electronics embedded software, respecting the requirements from that narrow domain.

Koala tries to achieve a strict separation between component- and configuration development. Component builders make no assumptions of the configurations in which the components are to be used. Likewise, configuration designers are not allowed to change the internals of components to suit their configuration [29]. This means that Koala follows the CBSE principle of separating component- and system development, and also the principle of using components as black-boxes.

Koala supports all stages of the lifecycle. Modeling is done using ADL like languages (CDL, IDL, DDL). Implementation is done in C. Packaging is supported through the use of repositories. Deployment is done at compilation. Particulars about each phase are given throughout this section.

Main entities of Koala are *components, interfaces, configurations, modules* and *switches*. Components are units of encapsulation, design, development and reuse. They communicate with their environment through interfaces. Components can be basic or compound – Koala has support for hierarchical binding, both through delegation and aggregation. The binding type is endogenous.

Interfaces are, as in for instance COM or OpenCOM, small sets of semantically related functions. Through interfaces a component: (i) provides its functionality to the environment, and (ii) requires functionality from the environment. Therefore, there is a distinction between *provides* and *requires* interfaces. The latter are similar to OpenCOM's receptacles (see Section 2.14). All requires interfaces must be bound to exactly one provides interface, and each provides interface can be bound to zero or more requires interfaces. All communication of a component with its context is routed through requires interfaces. This makes components to a large extent context-independent, as they rely on services only, rather than on specific servers (implementations of services) [30].

A component can have *optional interfaces*, both requires and provides. There is a function which tells if an optional requires interface is connected to a non-optional provides interface. A component must implement a function which tells if it actually implements an optional provides interface. Optional interfaces are modeled after COM's `QueryInterface` (see Section 2.4).

Although components should not contain configuration-specific information to be reusable, non trivial reusable components are allowed to be parametrized over all configuration-specific information. This is achieved by using requires interfaces – a component can require properties to be filled in by the configuration. Such interfaces are called *diversity interfaces*. Koala can express properties of inner components in terms of properties of outer components. These optional and diversity interfaces are distinctive features of Koala. The contractualization level of interfaces is syntactic. Interfaces are immutable, they cannot be changed once published.

A configuration is a set of components connected together to form a product, i.e. a system made from interconnected components. It is a top-level component with no interfaces on the border. Only configurations can be compiled and linked into executables [30].

A module is a unit of code, it represents the implementation of a basic component. Modules are also used to glue interfaces, as interfaces do not always fit perfectly. Modules are implemented in C. Since components communicate through C function calling, the interface type is operation-based, the interaction style is request-response, and the communication type is synchronous.

A switch allows to create bindings that depend on values of certain functions, thus enabling run-time reconfiguration of connections between components. A switch is equivalent to a conditional expression in a module.

Koala defines three languages:

- component definition language (CDL),
- interface definition language (IDL), and
- datatype definition language (DDL).

CDL describes the boundaries of a components. IDL describes an interface as a list of function prototypes in C syntax. DDL describes a datatype referring to other datatypes. An example of an interface definition, a basic component definition and a compound component definition is given in Table 6[10].

Table 6: An interface-, basic component- and compound component definition

Interface definition	Basic component definition	Compound component definition

`interface ITuner`
`{`
`void SetFrequency(int f);`
`int  GetFrequency(void);`
`}`
	`component CTunerDriver`
	`{`
	`provides ITuner ptun;`
	`  IInit  pini;`
	`requires II2c  ri2c;`
	`}`
		`component CTvPlatform`
		`{`
		`provides IProgram pprg;`
		`requires II2c slow, fast;`
		`contains`
		`  component CFrontEnd cfre;`
		`  component CTunerDriver ctun;`
		`connects`
		`  pprg  = cfre.pprg;`

10 Code examples from [29].

		cfre.rtun = ctun.ptun;
		ctun.ri2c = fast;
		}

Koala provides a tool (also named Koala) for component and configuration development. The tool uses partial evaluation techniques to optimize generated code – for instance unreachable components are not included, switch conditions that can be evaluated at compile time are turned into normal function calls.

Figure 5: Koala's graphical notation

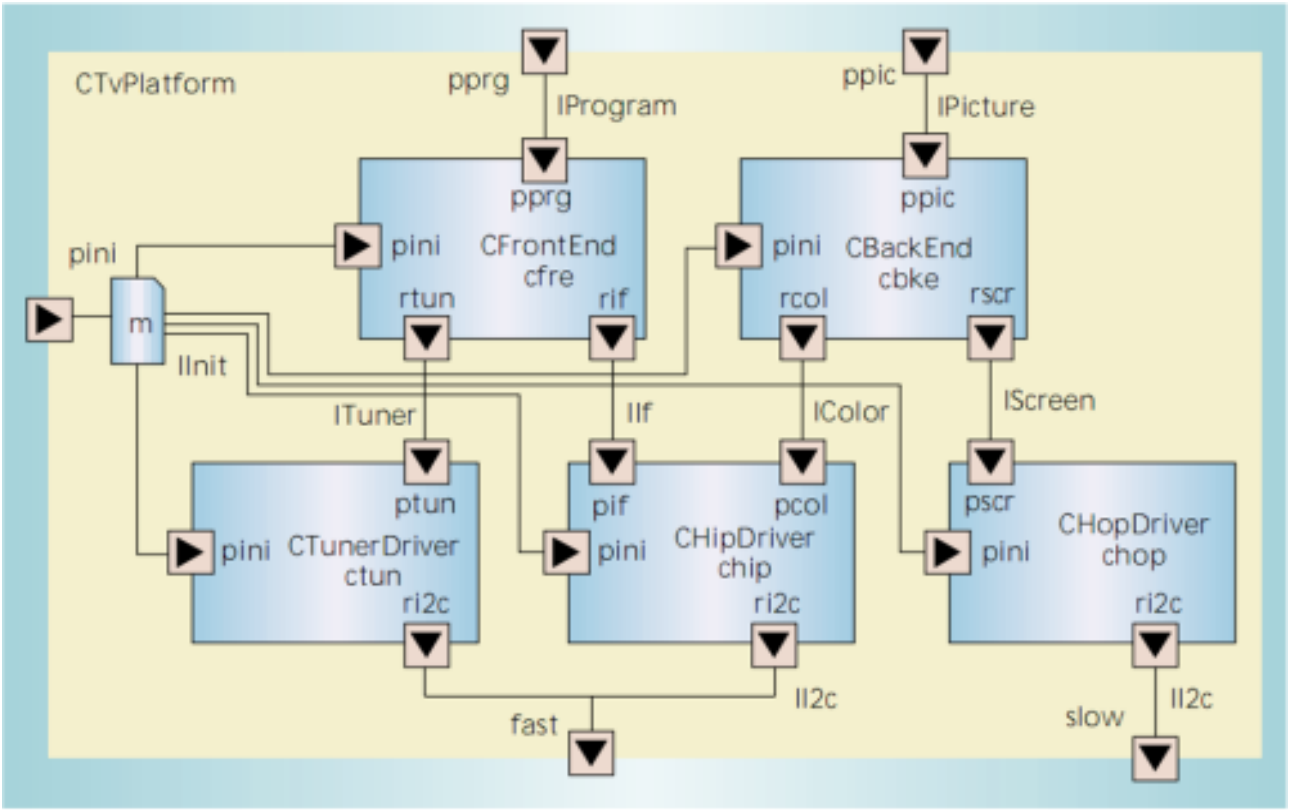

Figure 6: An example of a Koala configuration

The graphical notation of Koala is given in Figure 5 and a simple Koala configuration in Figure 6[11]. Components are deliberately made to resemble IC chips and configurations to look like electronic circuits. Interfaces look like pins of an IC chip. The triangles designate the direction of function calls.

Koala interfaces and Koala components are stored in an interface and a component repository, respectively. Changes to the interface repository can be made only after they have been approved by the interface management team. The following rules apply:

- Existing interface types cannot be changed.
- New interface types can be added.

Each component has a CDL description, a data sheet (a short document describing the component), and a set of C and header files. Changes to the component repository can only be made after approval by the architecture team. The following rules apply:

- New components can be added.
- An existing component can be given a new *provides* interface, but an existing *provides* interface cannot be deleted.
- An existing component can be given a new *requires* interface, but it must then be optional.
- An existing *requires* interface cannot be deleted, but it can be made optional [29].

In Koala there is a mechanism for specifying static EFPs, concretely resource usage (for instance static memory [31]). There is also support for compile-time checks of resources. Management of EFPs is endogenous system wide.

2.13 KobrA

KobrA (**KO**mponenten**B**asie**R**te **A**nwendungsentwicklung)[32] is a general-purpose software engineering method for the development of component-based application frameworks. It is also a hierarchical component model in which every component, regardless of its position in the hierarchy, is treated using the same set of concepts – a complete system is a component and every component, if it has appropriate properties, can be considered a system.

Components are described using text and UML models at two levels of abstraction:

- **Specification** – defines the components externally visible properties and behaviors which define the contract that the component fulfills.
- **Realization** – describes how the component fulfills its contract by using contracts with its sub-components.

Component interfaces are defined with UML models used by the specification of the component. Interface operations are defined on all three levels:

- Syntactic level – component operations are listed in the UML class diagrams with the distinction of optional operations.

11 Figure from [29].

- Semantic level – for each operation an operation schemata is defined, which shows the effects of the operation in terms of input parameters, changed variables, output values and pre- and post conditions.
- Behavior level – UML state-chart diagrams are used to describe how the component reacts to external stimuli.

Components are synchronously communicating via method calling (request-response).

No support is being provided for extra-functional properties.

KobrA augments the typical "binary-module" view of the component with a UML-based model, in a way enabling that the analysis and design activities are component-oriented and that the description of the core structure and behaviors are independent of the chosen component implementation technology.

KobrA method is composed of the two sets of activities. In the first set of activities the goal is to generate textual and UML-like component descriptions, and they are the following:

- Framework engineering activity – developing a generic framework that contains all the functionality of a family of applications which results with framework models that have certain variabilities due to differences between applications in the family of applications.
- Application engineering activity – all the variabilities in the framework models are resolved and in that way transforming them to application models.

The goal of the second set of activities is to generate an executable of the application, and they are:

- Implementation activity – maps the instantiated UML models to an executable representation.
- Build activity – creates binary modules that can be deployed in a target environment.

2.14 OpenCOM

OpenCOM is a component model developed at Lancaster University. OpenCOM v1 was originally described in [33], but has since then had a number of revisions listed in [34]. Currently OpenCOM v2 [35], [36] is under development. In this report we give an overview of OpenCOM v2.

OpenCOM v1 was based on a subset of Microsoft's COM, however, OpenCOM v2 removes its reliance from COM, as it aims to be truly platform independent. It is a general-purpose component model for construction of low-level systems software, such as embedded systems, operating systems, communications systems, programmable networking environments or middleware platforms. The goal of OpenCOM[12] is to support the unique requirements of a wide range of target domains and deployment environments. It tries to address target domain independence, deployment environment independence and negligible overhead. This is achieved by splitting the programming model into a simple, efficient and minimal kernel, and then providing on top of the kernel a set of extension mechanisms that allow the necessary tailoring [36].

12 From now on when saying OpenCOM we mean OpenCOM v2.

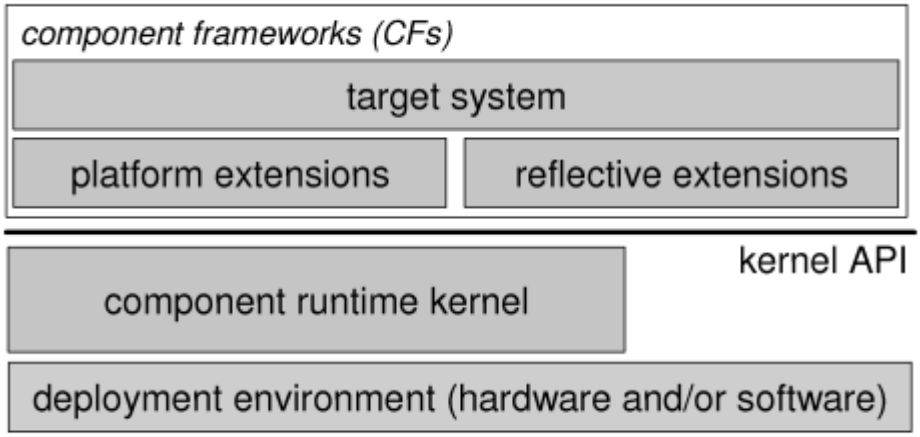

Figure 7: Overall OpenCOM architecture

The OpenCOM architecture is shown in Figure 7. Immediately above the deployment environment lies the *component runtime kernel* that supports basic services of loading and binding components. It is policy free, and its APIs are target system independent and deployment environment independent. For static systems it is used for initial configuration after which it can be unloaded so that it does not consume any resources. For dynamic systems it continues to exist at runtime. Above the kernel is the *extensions* layer, which enhances the basic kernel's loading and binding mechanisms in accordance with the target domain and deployment environment needs. One of the key architectural features of OpenCOM are *component frameworks* (CFs). A component framework is a tightly coupled set of components that:

- cooperates to address some focused area of concern,
- provides a well-defined extension protocol that accepts additional “plug-in” components that modify or extend the CF’s behavior, and
- constrains how these plug-ins may be organized [36].

Main entities of the OpenCOM programming model at the kernel level are *capsules*, *components* and *interaction points*. Capsules are containing entities into which components are loaded, instantiated and composed. Each capsule has a single kernel instance which offers the kernel's run time API used for dynamic loading and dynamic linking of components. Components are encapsulated units of functionality which interact with other components in their containing capsule exclusively through interaction points. Component types are templates from which component instances can be instantiated at runtime. There are two types of interaction points, *interfaces* and *receptacles*. Interfaces are units of service provision offered by components, and are defined as sets of operation signatures and associated data types. Receptacles are “required interfaces” that make explicit the dependencies of a component on other components. Receptacles support third-party deployment, as one can tell by looking at a components receptacles which other components must be present to satisfy the component's dependencies [36].

For definition of interfaces, receptacles and component types Object Management Group's Interface Definition Language (OMG IDL) is used. This programming model at the kernel level has been realized in C, C++ and Java. An IDL compiler is used to generate language specific representations

of the entities (component types, component instances, interfaces and receptacles) and glue code.

On the extensions level the main entities are *caplets*, *loaders* and *binders*. They are supported by a CF called the Platform Extensions CF. Caplets are specialized component-support environments that can be dynamically instantiated within a capsule. There are three main motivations for caplets. The first is for different caplets to represent different technology domains in the deployment environment. The second motivation is to use caplets to provide privacy and isolation between components of different privileges. The final motivation is to support heterogeneous component styles, i.e. different implementations of the abstract component concept. Loaders are used to load component types and instantiate component instances into a caplet in some particular manner. For instance a loader can perform security checks on the component types it loads. Binders are used for creating bindings between an interface and a receptacle, within and across caplets of a single capsule, in particular ways [36].

OpenCOM differentiates between two programmer roles:

- deployment environment programmer, and
- target system programmer.

The former creates caplets, loaders and binders for a particular deployment environment using the environment's native facilities. The latter develops target systems using the kernel's and Platform Extensions CF's API, together with the palette of caplets, loaders and binders provided by the deployment environment programmer.

OpenCOM provides support in the implementation and deployment stages of the lifecycle, while the modeling and packaging stages are not handled. Implementation is language independent. Deployment is done at run-time.

The following remarks apply only to the kernel level of OpenCOM. Remarks for the extensions layer cannon be given, as they depend on a particular implementation of caplets, loaders and binders.

Interface type is operation-based. There is a distinction between the provides- and requires parts, through OpenCOM's interfaces and receptacles. The contractualization level of interfaces is syntactic.

The interaction style is request-response, and the communication type is synchronous. The binding type is exogenous, as the interface of one and receptacle of another component are binded in a third component. There is no support for hierarchical binding.

There is no support for EFP specification or composition and analysis. Management of EFPs is endogenous per collaboration.

2.15 Palladio Component Model

Palladio Component Model (PCM) [37], [38], [39] is a domain-specific component model designed to enable early performance predictions for component-based software architectures of business information systems. Development of the model started in 2003 at the University of Oldenburg and is since 2006 continued at the University of Karlsruhe. Currently the model is at version 3.0 and is used both in industrial and academia projects. It is defined as a meta-model specified in Ecore [40].

An integrated modeling environment based on Eclipse RCP [41], called PCM-bench, is currently being developed, now being in the prototype phase. It enables creating PCM models using graphical editors, and then deriving performance metrics from these models using analytical techniques and simulation.

Two key features of PCM are:

- parameterized component quality-of-service (QoS) specification, and
- developer role concept.

The former is a special QoS specification for software components, which is parameterized over environmental influences that are unknown to component developers during component- design and implementation. This specification is called *resource demanding service effect specifications* (RDSEFF). Regarding the latter, PCM is aligned to different roles involved in component-based development and as such distinguishes between the following roles:

1. component developer,
2. system architect,
3. system deployer, and
4. domain expert.

RDSEFFs and particular roles are detailed in the following text. First a brief overview of all roles is presented in the following paragraph, than particulars about each role are given, with emphasis on the component developer.

A component developer specifies the functional- and extra-functional properties of components, which results in a *repository model*. A system architect assembles component specifications to form an *assembly model*. A system deployer specifies the resource environment to which the system will be deployed, providing a *resource environment model*. The system deployer also models the allocation of components from the assembly model to different resources of the resource environment, resulting in an *allocation model*. A domain expert is familiar with users of the system, and provides a *usage model* describing usage scenarios. Each role has its own domain-specific modeling language. From these partial models[13], a model of the complete system can be derived and then analyzed in terms of performance using multiple analysis methods, such as queuing networks, stochastic regular expressions or stochastic process algebra.

Regarding the lifecycle, the emphasis is on modeling, as seen above. However, the effort spent on creating a model of a system should be preserved when implementing it. In that sense a model-to-text transformation generates code skeletons from PCM models. The implementation uses either POJOs (Plain Old Java Objects) of EJBs (Enterprise JavaBeans). Deployment is done at run-time, which is inherent from EJB.

Component developer

In repository models *components* and *interfaces* are first class entities. Interfaces contain a list of service *signatures*, thus interfaces are operation-based. A signature has a name, a list of parameters, a return type and a list of exceptions it can raise during its execution. Its syntax is based on Object

13 In this overview we don't go deep into technical aspects of the models. Details can be found in the referenced literature.

Management Group's Interface Definition Language (OMG IDL). Interfaces have optional *protocols* and optional RDSEFFs (mentioned above, explained later). Protocols specify the order in which services are called, and can be modeled with various formalisms (finite state machines, regular expressions, Petri nets...). Signatures provide the syntactic interface contractualization level, RDSEFFS provide the semantic level and protocols provide the behavior level. Palladio supports interface inheritance.

Components either require or provide an interface, and implement services specified in their provided interfaces by using services specified in their required interfaces. Components can be either *basic* or *composite*. Composite hierarchies can be of arbitrary depth. Inner components are connected using *assembly connectors*. An assembly connector binds a provided role of one component with a required role of another component. *Delegation connectors* connect provided/required roles of composite components with provided/required roles of inner components. Thus, Palladio has support for hierarchical binding through delegation, and the binding type is exogenous.

The interaction style in Palladio is request-response and the communication type is synchronous.

Management of EFPs is endogenous system-wide. RDSEFFs abstractly model the externally observable behavior of a component. They specify: how a provided service calls the required services of a component, resource usage, transition probabilities, loop iteration numbers and parameter dependencies, all this to allow accurate performance predictions. RDSEFFs can be considered as a domain-specific modeling language which the component developer uses to specify performance related information for components. They represent the gray-box view of components.

Software architect

Software architects put components in so called *assembly contexts*, which are representations of component instances. Assembly contexts are connected using *system assembly connectors*. A system assembly connector binds a required role of a component in a given assembly context with a provided role of a component in a different assembly context. A set of connected assembly contexts is called an *assembly*[14], which is part of a *system*. Every system has exactly one assembly. A system exposes *system provided-* and *system required roles*. *System delegation connector*s bind system roles with the roles of the system's inner components. System provided roles are used by domain experts to model the usage of the system. System required roles model external services, which are not considered by the system architect to be part of the system.

System deployer

Palladio distinguishes between *active-*, *passive-* and *linking resources*. Active resources (eg. CPU, hard disk, memory) can execute requests, while passive (eg. threads, semaphores) can only be acquired and released. System deployers group active- and passive resources in *resource containers*. Resource containers are connected by linking resources. *Allocation contexts* are used to specify that a resource container executes an assembly context.

Domain expert

Domain experts specify user behavior using control flow constructs such as *sequence*, *alternative* and *loop*. For alternatives they specify branching probabilities and for loops they specify the

14 An assembly is different from a composite component in its visibility for the system deployer, as he can see the internals of an assembly, but cannot see the internals of a composite component.

number of iterations. Additionally, domain experts can specify the user workload and parameter values.

2.16 Pecos

The PECOS project [42], [43], [44], [45] provides a component model, a composition language and tools for development of software for field devices[15].

Life cycle

In the PECOS project defined a new language, named *CoCo*, for describing components and their compositions. Behavior of such compositions of components can be further modeled using Petri nets. PECOS currently supports specification of component implementation in C++ and Java programming languages. Within the project, tools for automatic generation of stub code for components and application for these two languages have been developed. Once compiled, components, and their compositions, are packed in Jar or DLL executable files. Inherently, PECOS components can only be deployed at compile time.

Constructs

The PECOS component model defines three types of components:

- *Active components*, which have their own thread of execution. They are used for performing long-term activities. PECOS systems are always modeled as active components whose behavior is defined by a composition of components.
- *Event components*. These components components have their own thread of execution, but do not model an on-going activity. Instead, their functionality is triggered by events. An example would be components that encapsulate hardware elements that generate periodical events.
- *Passive components* that execute synchronously, and their execution is scheduled by their active parent components.

Interfaces of PECOS components are described by the CoCo language. Interfaces constitute of ports that can be specified as either input or output, or both. Besides their direction, ports are defined by their name and data type. Additionally, properties like default values or value ranges for ports can be defined. Properties definition can include any functional or non-functional properties of components, ports or connections. Properties can also be grouped into property bundles which can then be assigned to multiple components or other CoCo architectural elements. Behavior of the components can be defined using Peri nets.

The CoCo language supports the concept of component inheritance. One component can inherit interface, or even parts of behavior, of another component, and possibly extend it or make it more specific. To facilitate this, the notions of *abstract component* and *component roles* have been introduced. An abstract components serves as a template for a system, leaving placeholders named component roles. Component roles define the interfaces that components need to implement by actual components that will be used in the implementation of the abstract components. This enables specification of architectural style for a family components or the entire applications.

15 A field device is an embedded system often used in the area of process control. It uses sensors to continuously gather data, analyses and reconciles this data, and reacts by controlling actuators [42].

Component in a data-flow (i.e. pipe and filter) manner, exchanging data through their ports. Special semantics is given to the ports of active and event components, because these types of components execute in their own threads. In these cases, synchronization between data used in different threads needs to take place for the composition to be predictable. In the C++ and Java implementation, these connections are realized using shared variables.

PECOS component model supports hierarchical component composition through the notion of composite components. A composite component consists of instances of components, and connections between those instances. Moreover, as already stated, all PECOS systems are modeled as active composite components. Ports of a composite component are defined by delegating some of the ports from its subcomponents to its outside interface.

Extra functional properties

Extra functional properties for components, their ports and connections can be defined using the CoCo property elements. Properties can be defined component description, but also as additional description of component instances when defining composite components.

Tools developed in the PECOS project also enable automatic generation of a Prolog knowledge base from the CoCo component definitions. This then enables testing of some properties of a component, or a composition of components, by using Prolog queries.

PECOS composite components are also associated with a scheduling definition for their subcomponents. Each subcomponent should have a worst-case execution time and the desired cycle time defined. Scheduling definitions also include order of execution for all subcomponents that a composite component consists of. PECOS supports both manual definition of scheduling or generation of scheduling definitions using information about component composition and properties of components. It is also possible to verify the scheduler using Petri nets.

2.17 Pin

Pin [46] [47] is a simple component technology developed at Carnegie Mellon Software Engineering Institute. Its purpose is to be used as a basis for prediction-enabled component technologies (PECTs).

Pin software components consist of prefabricated containers that provide a standardized interface, and custom code that they implement internally. Pin systems are modeled using ADL-like construction and composition language (CCL). Their implementation is defined by the C programming language, and such software components are packaged as .DLL files. Services like timers, interrupts, input devices, shared resources, process scheduling and intercomponent communication are provided to the components by the component run-time environment. Pin systems, called assemblies, are automatically generated and compiled to executable programs from the CCL specifications. Once compiled, Pin assemblies are fixed and the notion of internal components is lost.

Interfaces of Pin components are defined by *pins*. Pin component technology distinguishes between input pins, *sink pins*, and output pins, *source pins*. Sink pins denote incoming events to a component, and source pins denote outgoing events or procedure calls. Interfaces of components are specified by CCL. CCL supports specifying the syntax of the interface, but also the behavior of the component. Behavior of a component is described by *reactions*, which specify the relations

between the stimulus on sink pins and possible responses on output pins.

Software components in Pin component technology can interact only through their sink and source pins. This interaction can be either synchronous (procedure call) or asynchronous (event-based). Connections between sink and source pins are realized using *connectors*. Besides the definition of interconnection of pins, connectors can also have connection policies. An example policy would be the size of an input buffer. Units of component composition in Pin are called *assemblies*. Binding of the components is endogenous. Pin does not support hierarchical composition of component assemblies.

Pin components expose their *analytical interface* by specifying *properties* for components, assemblies, pins, reactions and the environment. Properties are n-tuples consisting of a name, value, and additional property-specific information (e.g. confidence interval of the property value). These properties can be used by a reasoning framework to predict properties of component compositions. PECT currently supports three reasoning frameworks for Pin component model: λ_{ABA} – for predicting average latency in assemblies with periodic tasks, λ_{SS} – for predicting average latency in stochastic tasks managed by a sporadic server, and ComFoRT – for formal verification of temporal safety and liveness.

2.18 ProCom

ProCom [48] is a component model for control-intensive distributed embedded systems and is designed to cover the whole development process in the vehicular-, automation- and telecommunication domains.

Typically, complex distributed embedded systems from targeted domains have different characteristics at different levels of granularity. ProCom tackles this problem by using two layers: ProSys and ProSave.

ProSys is a hierarchical component model which acts as an upper layer that models the system as a number of active and concurrent subsystems which communicate by asynchronous message passing.

ProSys subsystems can be: composite subsystems, subsystems realized with ProSave or wrapped legacy subsystems. Each subsystem is specified by:

- Typed input- and output ports which express what messages the subsystem receives and sends. Message ports are connected with message channels which support n-to-n communication.
- Attributes and models related to functionality, reliability, timing and resource usage.

ProSave is the lower layer which models the internal design of a single ProSys subsystem down to primitive functional components implemented by code.

ProSave components are passive, reusable units of functionality that can either be realized by code (C functions), or by interconnected sub-components. They use pipe-and-filter communication paradigm and are typically not distinguishable as individual units in the final executing system.

The architectural style is based on a data/control flow model with a separation of data transfer and control flow, which is manifested with the existence of data- (enable data read, write) and trigger

ports (control the activation of components).

Information about a component is represented using structured attributes and its functionality is made available by a set of services. Attributes define simple or complex types of component properties such as behavioral models, resource models, dependability measures and documentation. Each service consists of:

- *Input port group* – contains a trigger port for activation and a set of data ports for required data.
- *Output port groups* – contains a set of data ports and a trigger port which indicates when new data is available.

Components can be connected using:

- Simple connections that connect two ports and that can be used to transfer data or control.
- Connectors – constructs that may be used to control the data- and control-flow, for example to fork or join data or trigger connections.

Components and information about them (requirements, behavior models, resource usage) are stored in a file system based repository.

Connecting ProSave and ProSys

ProSys subsystems can be defined with a collection of interconnected ProSave components, ProSave connectors, and additional connector types such as:

- Input message port which acts as a ProSave connector with one output trigger port and one output data port. Whenever a ProSys subsystem receives a message, the message port writes message data to the output port and activates the output trigger.
- Output message port is similar to the input message port only it has one input trigger and one input data port. When a trigger is received it sends a message with the data from the data port.
- Clock is used for generating periodic triggers. It only has one output port which is periodically triggered.

2.19 Robocop

Robocop [49] [50] is a project that defines an open middleware layer for high volume consumer electronics. It aims to support definition, modeling and trading of software components, their use in consumer electronics applications, and run-time upgrades and reconfiguration of such applications.

Life cycle

Robocop defines components as units of trade. A Robocop component is a set of different models that are related to each other. These models address different aspects of the component that can be of interest to different stakeholders. An example of such models are interface definition, behavioral models, resource consumption models, etc. Robocop is open in the way that it does not limit the number or types of models that a component consists of. A special type of model is the *executable component*. Executable component is a binary representation of the component that implements its

functionality and can be executed. A Robocop component can contain multiple executable components that are targeted for different platforms or operating systems.

Robocop architecture defines three frameworks that support different concerns of component's life cycle.

The Development framework defines how different stakeholders (e.g. component vendors, system integrators) relate to each other, to the component model and entities like component repositories, and target devices.

The Run-time framework defines a partial architecture for Robocop devices. It consists of the Robocop component model and the *Robocop Runtime Environment* (RRE). To achieve a minimal resource footprint, standard RRE supports only registration of components and services, and location and instantiation of services. Robocop component model defines a standardized part of service interface that, together with RRE, provides mechanisms for run-time binding of components and reconfiguration of Robocop systems.

The Download framework enables run-time upgrades and extension applications built with Robocop technology. This is achieved by providing mechanisms for locating new components, testing if they are suitable for use in a given system, and transfer of new components from repositories to target devices.

Constructs

The functionality of executable Robocop components is encapsulated in *services*. To expose that functionality, services define *interfaces* they provide. Interfaces are groups of semantically related named operations. Services also explicitly expose their dependencies on functionality from other components through their required interfaces.

One service can define that it is *compliant* with another service. When one service defines that it is compliant with another, it indicates that it provides at least the same interfaces that the latter service provides, and that it requires the same interface that the latter service requires.

Concept of interface inheritance is supported. When one (derived) interface inherits another (base) interface, it indicates that the derived interface supports all operations that the base interface defines, and that those operations have the same semantics in both interfaces. Only single interface inheritance is supported, but the depth level of nesting is not limited.

Service and interface definitions are defined using *Robocop IDL* (RIDL), an interface definition language derived from Corba IDL. Although RIDL definitions provide only syntactical information, semantics and behavior of components can be described by other models that the component consists of.

Robocop services support reflective behavior, in the sense that they can dynamically discover interfaces that a service supports.

Services interact with each other in a client-server manner, in which a client service calls methods that the server service provides through its interfaces.

Information about binding between required and provided interfaces of different services is stored in the services themselves. Standard service management interface that each service must expose provides mechanisms for interface binding management. This enables dynamic and run-time configuration of assemblies of components.

Robocop does not provide any means for hierarchical composition of components.

Extra functional properties

Extra functional properties of Robocop components can be given in models that the component consists of. These extra functional models can include timeliness, reliability, safety, security and resource consumption.

Robocop implements resource management through the *Resource management framework*. The aim of this framework is to prevent resource overloads on embedded devices that support dynamic updates or upgrades. It introduces a notion of *resource-aware consumers*, which are application entities that have information about resources needed for its operation. A special type of such entities are the *quality-aware consumers*, which consume different amount of resources depending on the level of quality they provide in a given moment. The consumers can register their resource needs to the framework, which can then guarantee them requested resources or deny their request. The framework can also optimize system quality depending on the available resources.

A solution of memory consumption of Robocop applications is given in [51].

2.20 Rubus Component Model

The Rubus concept [52] is a collection of methods and tools from Arcticus Systems for development of dependable embedded real-time systems. It was introduced for industrial use in 1996, but has since then evolved, in cooperation with various partners of Arcticus Systems, both from industry and academia. The Rubus concept encompasses (among other) the Rubus Component Model, the Rubus Integrated Component Environment (an IDE for component-based development of real-time systems) and Rubus OS (a real-time OS designed for dependable real-time systems).

The Rubus Component Model[16] is developed in cooperation with Mälardalen University. It is currently in version 3 [53], and is intended for development of distributed, resource-constrained, embedded control systems, with a mix of hard-, soft- and non real-time requirements. Thus it is a domain-specific component model. It aims at supporting an overall high-level descriptive view of the system functionality, focusing on three important activities in real-time development – design, analysis and synthesis. The model supports all stages of the lifecycle. Modeling is done using Rubus Design Language, a non standard modeling language with an intuitive graphical representation. Implementation is done in C. Rubus components are packaged in a file system based repository. Deployment is done at compilation, and Rubus components are executed as tasks of Rubus OS.

Architectural elements of Rubus are collectively named *software items* (SWIs). Basic Rubus components are called *software circuits* (SWCs). Each SWC is defined by its behavior, internal state data and interface. An SWC can have multiple behaviors, each one represented by a specific C entry function. Internal state data is used to preserve data across multiple executions.

Rubus interfaces are port-based. There is a distinction between the provides- and requires part, through *output*- and *input ports*, respectively. Each port has a direction to symbolize in which way the signal flows. Interfaces are specified as C header files, and the contractualization level is syntactic. As distinctive features Rubus has *data*- and *trigger ports*, which capture data- and control flow, respectively. The distinction between data- and trigger ports makes data access and

16 From now on when saying Rubus, we mean Rubus Component Model.

synchronization explicitly visible from the design. Data ports are typed. Trigger output ports can be *unconditional* and *conditional*. The former always produce a trigger signal after the execution of a SWC, while the latter may or may not produce a trigger signal after execution, depending on conditions within the SWC.

The run-cycle of an SWC is the following: idle → ready → copy input → execute → produce output → terminate → idle. The SWC becomes eligible for execution when it receives a signal at its trigger input port. Before execution data input port values are read. After execution the SWC produces data- and trigger signals at its output ports and then returns to idle state.

Assemblies (ASMs) and *composites* (CMPs) provide the means for hierarchical binding through delegation. They have sets of input- and output ports whose signals are delegated to ports of internal SWIs. ASMs and CMPs give structure and abstraction, but provide no semantics and can in that sense be flattened to contain only SWCs. CMPs can be divided as parts of it can be deployed on different nodes, whereas an ASMs is undividable and can only be deployed as a whole on a single node.

The binding type in Rubus is endogenous. The interaction style is pipe-and-filter and the communication type is synchronous.

A s*ystem* is a top level hierarchical entity that describes the software logic and the software architecture (not the hardware architecture showing which SWIs execute on which hardware nodes) of a complete distributed system. At run-time a system executes in one out of a set of predefined *modes*, and can make transitions (*mode shifts*) from one mode to another. Modes are means to distinguish different states or conditions of a system – for instance a system executes a certain type of functionality during start-up, another type during normal operation, and a third type during errors.

Rubus also defines several SWIs for data and triggering [53]: *source items*, *sink items*, *named data items, clock items* (generate periodic triggering), *interrupt and event items* (generate sporadic triggering), *down sampling items* and *precedence items*.

External devices, such as sensors and actuators have no special status in Rubus and are treated as other components. Sensors are modeled by SWCs without any data input ports and with at least one trigger input port, while actuators are modeled by SWCs without any data output ports.

Management of EFPs in Rubus is endogenous system wide. Timing properties of SWCs and real-time requirements on the execution can be specified. Regarding the former, to enable timing analysis at design time, each SWC is associated with a run-time profile describing its run-time properties on different platforms. The latter are specified within the context of an ASM/CMP as bounds on time from the generation of a trigger signal to the generation of another trigger signal.

2.21 SaveCCM

SaveCCM (SaveComp Component Model) [54] is a research, domain-specific component model developed at Mälardalen University. It is intended for embedded control applications for vehicular systems, mainly considering the safety-critical subsystems responsible for controlling vehicle dynamics (such as power-train, steering, braking, etc.). It is a simple component model that limits the flexibility of modeling to enable analyzability with respect to timing.

SaveCCM systems are developed using a custom development environment called Save-IDE [55], which is implemented in the form of a plugin for the Eclipse platform [27]. SaveCCM provides support for all stages of the lifecycle. System architecture modeling is done using the SaveCCM graphical language (Figure 8), which is similar to UML component diagrams [56]. From the graphical description of a system, Save-IDE can generate its textual representation in XML adhering to the SaveCCM DTD. Component behavior modeling is done using timed automata extended with tasks [57]. SaveCCM has two realizations, one by transforming components to real-time operating system tasks [58] and a realization in JavaBeans [59]. In the former case components are implemented in C and packaged in a file system based repository, and in the latter case components are implemented in Java and packaged as JAR packages. In both realizations deployment is done at compilation.

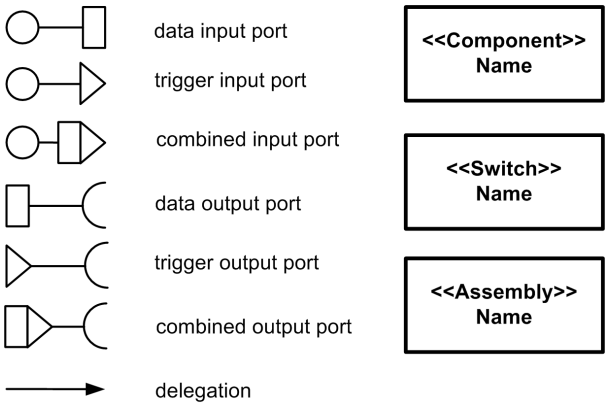

Figure 8: SaveCCM graphical notation

The main architectural elements of SaveCCM are *components*, *switches* and *assemblies*. The interface of an architectural element is defined by a set of *input*- and *output ports*, thus the interface type is port-based and there is a distinction between the requires- and provides part. Interfaces are specified in XML adhering to the SaveCCM DTD. The interface contractualization level is syntactic. The interaction style is pipe-and-filter and the communication type is synchronous.

As distinctive features SaveCCM has *data*-, *trigger*- and *combined ports*. Data ports capture data transfer, while trigger ports capture control flow. Data ports are typed and have overwrite semantics, and only data ports of matching types can be connected. Combined ports have both triggering- and data functionality, but semantically a combined port is equivalent to one trigger port and one data port.

In addition to ports, the interface of a component contains a series of quality attributes, for instance worst case execution times, reliability estimates etc. Each attribute is associated with a value and possibly a confidence measure. The quality attributes are used for analysis.

Components represent basic units of encapsulated behavior. The functionality of a component is typically defined by an entry function. These are *basic components*. However, there are also *composite components*, for which the functionality is defined by an internal composition of subcomponents (and possibly delays and switches, described below). Subcomponents can be basic components or again composite components (thus creating a hierarchy of arbitrary depth). The

hierarchical binding is done by delegation. The binding type is endogenous.

A component is initially idle and remains in that state until all its trigger input ports are activated. At that point it goes to active state, i.e. it has been triggered. This initiates the read phase, in which the data input port values are stored internally, to ensure consistent computation. Next is the execute phase, in which computations are performed. Then comes the write phase, in which data is written to the data output ports. Finally, the input triggers are deactivated and the output triggers are activated, before the component returns to idle state. The strict "read-execute-write" semantics ensures that once a component is triggered, the execution is functionally independent of any concurrent activity.

There are two additional types of components, *clock* and *delay*, which are in charge of manipulating trigger signals. A clock is a trigger generator, while a delay detains a trigger signal for a certain amount of time.

Switches enable dynamic modification of the connections between components by providing means for conditional transfer of data and/or triggering. A switch consists of a number of conditional mappings between an input- and an output port of the switch (either data or trigger). Each mapping is guarded by a logical expression. If this expression evaluates to true, the mapping holds, otherwise it is broken. Data input ports of a switch which are part of such a logical expression are called *setports*. Switches are not triggered, they respond immediately to arrival of data- or trigger signals at their input ports, and relay them according to the current mappings.

Assemblies are encapsulated subsystems. As an assembly can break the "read-execute-write"' semantics, it should only be viewed as a mechanism for naming a collection of components and hiding the internal structure, rather than a mechanism for component composition.

External devices, such as sensors and actuators should not be modeled by SaveCCM. Their presence is seen through the use of *external ports*. An external port is a port that is not connected with any other port in the model, but has a label mapping it to some external entity.

The SaveCCM semantics is formally defined by a two step transformation – first from the full SaveCCM language to a similar but simpler language called SaveCCM Core, and then into timed automata with tasks. SaveCCM Core is a minimal language consisting of three elements (basic component, composite component, conditional connection) with which all constructs of the full SaveCCM can be described.

Management of EFPs in SaveCCM is endogenous system wide. Specification of timing properties is supported, but aforementioned attributes enable generic specification of EFPs. Timing properties can be analyzed at design time using the UPPAAL Port model checker [60].

2.22 Sofa

Software Appliances – SOFA is a component model developed within the SOFA academic project at Charles University in Prague [61]. It encompasses several software domains, such as the communications middleware, component management, component design and security. Key issues addressed by SOFA include component transmission protocol, dynamic component downloading and updating, hierarchical top-down design, distributed deployment and versioning, and support for component trading and licensing.

The SOFA 2.0 [62]component model is an extension of the SOFA component model with several new services: dynamic reconfiguration, control interfaces and multiple communication styles between the components.

Components

In SOFA a system is built out of a set of dynamically updateable components. Every SOFA component is specified by its *frame* and *architecture*.

The *frame* provides a black box view of a component through the provided and required interfaces. Optionally, a behaviour protocol and component properties can be declared. Behaviour protocol is used to formally specify communication between SOFA components, while the properties are intended for parametrizing the component.

SOFA *architecture* is an implementation of a frame. Every frame can be implemented by more than one architecture. The architecture may be either composed or primitive. Composed architecture is built of other components – subcomponents with specified interconnections via interface ties. It provides a grey-box view of a component, as it describes the structure of a component until the first level of nesting in the component hierarchy. A primitive architecture contains no subcomponents, only a code implementing the component's functionality described by the frame.

When defining internal structure of a composed component, its subcomponents are specified by frames at a design time, At deployment time, architectures that implement the subcomponents are determined and instantiated. This separation of component's external view (the frame), from component's internal view (the architecture) is one of big advantages of SOFA component model. It allows use of component types (frames) at compile time, and specifying implementation of component functionality at deployment time.

Interfaces

SOFA components and systems are specified by an ADL-like language, Component Description Language (CDL). CDL descriptions are used to describe provided and required interfaces, frames and architectures of SOFA components. Interfaces are based on CORBA IDL's interfaces with an extension to provide means to specify version and behaviour protocol of interfaces.

The resulting CDL is compiled by a SOFA CDL compiler to their implementation in a programming language C++ or Java, and then is stored in the Type Information Repository (TIR). TIR manages an evolution of component's description and can store several versions of every element. In the CDL descriptions, a developer can specify references to a concrete version of previously compiled types stored in TIR.

Connectors

SOFA components can be composed by connectors. Connectors separate interconnection semantics and deployment dependant details from an application logic placed in components. In SOFA, connectors are first-class entities like components. Each type of connector implements semantics of specific type of interaction, and are similarly to components, specified by connector frame and connector architecture. Connector frame defines the type of a connector by describing services provided by a connector. Connector architecture describes connector internals and can be *simple* (containing only primitive elements, directly implementing connector frame) or *compound* (containing instances of other connectors or components).

There are three predefined connectors in SOFA: *CSProcCall* (for synchronous calls), *EventDelivery* (for asynchronous calls) and *DataStream* (for data streaming).

In addition to connector types, connectors can be distinguished depending on entities they connect:

- bind – connects required interface to provided interface on the same hierarchy level.
- delegate – forwards requests from provided interface of a component to provided interface of its subcomponent.
- subsume – passes requests from a subcomponent's required interface to a required interface of a component.

SOFANode

SOFANode is an environment for developing, distributing and running SOFA applications. It consists of several logical parts:

- Template repository – which contains CDL descriptions as well as implementations of components
- CDL Compiler, Template Information Repository and Code generator – for application development
- RUN – the runtime environment for launching component applications

SOFANode can be distributed across a network over several hosts. Several SOFANodes connected form a SOFANet.

References

[1] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, Michel Chaudron, A Classification Framework for Component Models, submitted for IEEE Transactions on Software Engineering, 2008

[2] AUTOSAR Development Partnership, AUTOSAR - Technical Overview, 2008

[3] Heinecke H., Damm W., Josko B., Metzner A., Kopetz H., Sangiovanni-Vincentelli A., Di Natale M., Software Components for Reliable Automotive Systems, Design, Automation and Test in Europe, 2008

[4] AUTOSAR Development Partnership, AUTOSAR - Software Component Template v3.0.1, 2008

[5] A. Basu, M. Bozga, J.Sifakis, Modeling Heterogeneous Real-time Components in BIP

[6] Ji Eun Kim, Rahul Kapoor, Martin Herrmann, Jochen Haerdtlein, Franz Grzeschniok, Peter Lutz, Software Behavior Description of Real-Time Embedded Systems in Component Based Software Development, ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, 2008

[7] Microsoft, COM: Component Object Model Technologies, http://www.microsoft.com/com/default.mspx

[8] Microsoft, MSDN: COM (Component Object Model), http://msdn.microsoft.com/en-us/library/ms680573.aspx

[9] Sara Willliams, Charlie Kindel, The Component Object Model: A Technical Overview, http://msdn.microsoft.com/en-us/library/ms809980.aspx, 1994

[10] Ivica Crnković, Magnus Larsson, Building Reliable Component-Based Software Systems, Artech house, 2002

[11] Ke Xu, Sierszecki Krzysztof, Angelov Christo, COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems, RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007

[12] Ke Xu, Pettersson Paul, Sierszecki Krzysztof, Angelov Christo, Verification of COMDES-II Systems Using UPPAAL with Model Transformation, RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008

[13] Rémi Bastide, Eric Barboni, Amélie Schyn, Component-Based BehaviouralModelling with High-Level Petri Nets, Third Workshop on Modellingof Objects, Components and Agents (MOCA'04), 2004

[14] Fintan Bolton, Pure CORBA, Sams Publishing, 2001

[15] OMG, CORBA v4.0, 2006

[16] Sun Microsystems Inc., Enterprise Java Beans 3.0, Final Relase, 2006

[17] Sun Microsystems, JavaBeans,

http://java.sun.com/javase/technologies/desktop/javabeans/

[18] Sun Microsystems Inc., Java Persistence API, http://java.sun.com/javaee/technologies/persistence.jsp

[19] Sun Microsystemc Inc., Remote Method Invocation, http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[20] The ObjectWeb Consortium, The Fractal project web pages, http://fractal.ow2.org/,

[21] The ObjectWeb Consortium, Think web pages, http://think.ow2.org/,

[22] The ObjectWeb Consortium, Juila web pages, http://fractal.ow2.org/julia/,

[23] E. Bruneton (France Telecom R&D), T. Coupaye (France Telecom R&D), J.B. Stefani (INRIA), The Fractal Component Model Specification, 2004

[24] Sünder C., Zoitl A., Christensen J. H., Steininger H., Fritsche J., Considering IEC 61131-3 and IEC 61499 in the context of Component Frameworks, The IEEE International Conference on Industrial Informatics, 2008

[25] Rockwell Automation, Function Block Development Kit, http://www.holobloc.com/fbdk/README.htm,

[26] Thramboulids K., Doukas G., Frantzis A., Towards an Implementation Model for Function Block-Based Reconfigurable Distributed Control Applications, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2004

[27] Eclipse, http://www.eclipse.org/

[28] Sun Microsystems, NetBeans, http://www.netbeans.org/

[29] Rob van Ommering, Frank van der Linden, Jeff Kramer, Jeff Magee, The Koala Component Model for Consumer Electronics Software, IEEE Computer, 2000

[30] Rob van Ommering, Building product populations with software components, Proceedings of the 24th international Conference on Software Engineering, 2002

[31] A.V. Fioukov, E.M. Eskenazi, D.K. Hammer, M.R.V. Chaudron, Evaluation of static properties for component-based architectures, Proceedings of 28th EUROMICRO conference, 2002

[32] C. Atkinson, J. Bayer, D. Muthig, Component-Based Product Line Development: The KobrA Approach,

[33] Michael Clarke, Gordon S. Blair, Geoff Coulson, Nikos Parlavantzas, An Efficient Component Model for the Construction of Adaptive Middleware, Proceedings IFIP Middleware, 2001

[34] Lancaster University, Overview of OpenCOM, http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/opencom.php

[35] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, Jo Ueyama, A Component Model for Building Systems Software, In Proceedings IASTED Software Engineering and Applications SEA'04, 2004

[36] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee,

Jo Ueyama, Thirunavukkarasu Sivaharan, A Generic Component Model for Building Systems Software, ACM Transactions on Computer Systems (TOCS), 2008

[37] University of Karlsruhe, The Palladio Component Model, http://sdqweb.ipd.uka.de/wiki/Palladio_Component_Model

[38] Ralf Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, Michael Kuperberg, The Palladio Component Model, Technical report, University of Karlsruhe, 2007

[39] Steffen Becker, Heiko Koziolek, Ralf Reussner, Model-Based Performance Prediction with the Palladio Component Model, Proceedings of the 6th International Workshop on Software and Performance, 2007

[40] Eclipse Modeling framework, http://www.eclipse.org/modeling/emf/?project=emf

[41] Eclipse RCP, http://wiki.eclipse.org/index.php/Rich_Client_Platform/

[42] Tomas Genssler, Alexander Christoph, Benedikt Schutz, Michael Winter, Chris M. Stich, Christian Zeidler, Peter Müller, Andreas Stelter, Oscar Nierstrasz, Stéphane Ducasse, Gabriela Arévalo, Roel Wuyts, Peng Liang, Bastiaan Schönhage, Reinier van der Born, PECOS in a Nutshell, 2002

[43] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P. Black, Peter O. Müller, Christian Zeidler, Thomas Genssler, Reinier van den Born, A Component Model for Field Devices, Lecture Notes in Computer Science, 2002

[44] Michael Winter, Thomas Genßler, Alexander Christoph, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Peter Müller, Chris Stich, Bastiaan Schönhage, Components for Embedded Software - The PECOS Approach, The Second International Workshop on Composition Languages, in conjunction with the 16th ECOOP, 2002

[45] R. Wuyts, S. Ducasse, Non-Functional Requirements in a Component Model for Embedded Systems, International Workshop on Specification and Verification of Component-Based Systems, OOPSLA

[46] Scott Hissam, James Ivers, Daniel Plakosh, Kurt C. Wallnau, Pin Component Technology (V1.0) and Its C Interface, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2005

[47] Kurt C. Walnau, Volume II: A Technology for Predictable Assembly from Certifiable Components, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2003

[48] T. Bureš, J. Carlson, S. Sentilles, A. Vulgarakis, ProCom - A Component Model for Distributed Embedded Systems

[49] H. Maaskant, A Robust Component Model for Consumer Electronic Products, Philips Research Book Series Volume 3

[50] J. Muskens, M.R. V Chaudron, J.J. Lukkien, A Component Framework fror Consumer Electronics Middleware, Lecture Notes in Computer Science, 2005

[51] M. de Jonge, J. Muskens, and M. Chaudron, Scenario-based prediction of run-time resource consumption in component-based software systems, In Proceedings: 6th ICSE

Workshop on Component Based Software Engineering: Automated Reasoning and Prediction, 2003

[52] Arcticus Systems, http://www.arcticus-systems.com/

[53] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, Kurt-Lennart Lundbäck, The Rubus Component Model for Resource Constrained Real-Time Systems, 3rd IEEE International Symposium on Industrial Embedded Systems, 2008

[54] Mikael Åkerholm, Jan Carlson, John Håkansson, Hans Hansson, Mikael Nolin, Thomas Nolte, Paul Pettersson, The SaveCCM Language Reference Manual, Technical report, Mälardalen University, 2007

[55] Séverine Sentilles, John Håkansson, Paul Pettersson, Ivica Crnković, Save-IDE – An Integrated development environment for building predictable component-based embedded systems, Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 2008

[56] Object Management Group, Unified Modeling Language, http://www.uml.org/

[57] Amnell, T., E. Fersman, L. Mokrushin, P. Pettersson and W. Yi, Times: a Tool for schedulability analysis and code generation of real-time systems, Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems, LNCS (2003), 2003

[58] Kathrin Dannmann, Synthesizing Real-Time Components to Run-Time Tasks, Master's Thesis, University of Oldenburg, 2009

[59] Juraj Feljan, Using JavaBeans to Realize a Domain Specific Component Model, submitted to Software Engineering and Advanced Applications (SEAA09), 2009

[60] Uppsala University, Aalborg University, UPPAAL Port, http://www.uppaal.com/port/

[61] SOFA 1 web page, http://sofa.ow2.org/sofa1/, 2009

[62] SOFA 2 web page, http://sofa.ow2.org/, 2009