UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

THESIS no. 1677

COMPONENT BEHAVIOR MODELING

Ivan Ferdelja

Zagreb, May 2009.

Table of contents:

1. Introduction

Requirements on software systems in all domains, from the smallest embedded systems, desktop and business software to large scale industrial systems are increasing constantly. Software development is thus faced with responding to huge challenges. Systems are expected to be more reliable, less resource hungry and brought to market as soon as possible. An approach to software system development based on the reusability principle and has an increasing popularity is component-based development (CBD), where systems are built by composing independent, tested and trusted software components. Component-based software engineering (CBSE) is a structured and a systematic approach to CBD whose purpose is to provide processes and technologies for successful mainstream application of CBD, which is expected in the future.

Considering that the component is the fundamental building block of software systems in CBD, it is important to investigate methods for implementing its functionality, i.e. the services it provides. Services provided by components are mostly algorithmic, i.e. they realize specific calculations on relatively simple sets of input data. Thus implementation is given as source code in a fast programming language, such as C. However, when implementing the functionality in a concrete programming language, the implementations correctness becomes harder to verify, and can mostly be achieved through testing. Component also becomes dependent on a specific programming language.

An approach that offers some solutions to these and other problems is behavior modeling. Modeling has today become a mainstream engineering technique, especially with the emergence of the Unified Modeling Language (UML). Although many arguments exist pro and contrary of the modeling approach, it is nevertheless reasonable to investigate possibilities of applying existing methods for modeling the behavior of software components. The situation in the field of behavior modeling languages is far from ideal. There is no clear definition of what a behavior modeling language is, nor do there seem to be any systematic attempts to generally structure the field of such languages.

The purpose of this thesis is to attempt to analyze and summarize behavior modeling languages and in particular two languages that have shown most promising, Executable UML (xUML) and Construction and composition language (CCL). Considering the situation in the field of such languages, a short summary of several discovered languages is given, but the majority of attention was given to xUML and CCL since CCL has proven to be successful in its domain and xUML represents an important family of modeling languages based on the ubiquitous UML. The secondary task was to provide a model for applying such behavior modeling languages for modeling the behavior of the ProCom component model, which has been developed as part of the PROGRESS project and aims at enabling component-based development of resource limited and control intensive embedded systems.

An overview of the CBSE concepts and the development process is given in chapter 2. Chapter 3 gives an overview of the ProCom model. CCL is analyzed in detail in chapter 4 and strategies for its application to ProCom are given in chapter 5. Development processes using xUML are covered in chapter 6 and the application of xUML to ProCom is the subject of chapter 7. Chapter 8 contains a short language summary of a few other discovered modeling languages. Conclusion in chapter 9 gives a final closure to the thesis.

2. Overview of CBSE

Software development today is experiencing a massive expansion into virtually all areas of human life, from industrial and business to home applications. Practically any new feature that a technical product will offer is mostly based on its software. Therefore requirements on software systems and thus on the development process have become very high. Requirements are high both from the viewpoint of shortening the development time, minimizing the cost of development, etc. Specialized systems also have special requirements, for example software for embedded systems has special requirements on minimizing the required resources etc.

Thus the production of software is more complex then ever before and software is becoming more ubiquitous. Software industries ability to build larger and more complex systems has improved considerably. Important public infrastructure relies on complex computer system. An incredible amount of different technologies exist for building software systems. However, when observing fundamental software development processes not much has changed. Waterfall model is still the dominant approach for development of systems, testing is still the dominant validation approach, etc. [14]

Therefore considerable problems still exist, which cause the majority of projects to fail to meet their deadline, budget and quality requirements.

One possible solution for problems of software development today is the reuse principle. Reusability is not a new idea but in spite of that it has not become the main development method. Component based development (CBD) is a developing new approach that is based on the idea of reusability. In CBD system are built from components already developed and prepared for integration. CBD therefore offers many advantages such as: effective management of complexity, reduced time to market, increased productivity, improved quality and a wider range of reusability. [1]

Component based software engineering (CBSE) is a structured and systematic approach aimed at providing processes and technologies for CBD. CBSE is based on the concept of a component. It also introduces other fundamental concepts such as: interface, framework, and model. These concepts will be put into perspective in the following sections. [1]

2.1. Components

A component is a reusable unit of deployment and composition [1]. Components are the fundamental building block of component based systems and must therefore be completely understood. However no consensus still exists about what a component really is, thus there are many different definitions. From a practical perspective component comprises many important parts that need to be considered and specified. Also the environment whose part the component will be must also be understood and specified.

Fundamental component characteristics can be summarized as: [14]

- Standardized – a component used in a CBSE process must conform to a component model which may define interfaces, composition and deployment rules etc.

- Independent – it must be possible to compose and deploy a component without explicitly requiring any other components

- Composable – all components interactions must occur through the components public interface and the component must provide information about it self

- Deployable – a component must be able to operate as a standalone entity on a component platform supporting the component model

- Documented – component must contain both its syntactical and semantical specification required for its possible usage

Viewed externally, component is defined through its interface that becomes the access point to the component considering that the component is a black-box, i.e. components internal implementation is not accessible from the outside. Practically, the black-box principle cannot always be achieved. Separation of the components implementation from its external access point is similar to the concept of encapsulation in object oriented development and it serves a similar purpose, i.e. the implementation can be changed without affecting the users of a component. Also it is possible to extend the interface without changing the implementation.

From a functional point of view, a component offers some services to its environment, i.e. to its users. Thus the components interface must specify the service that the component provides. Specification of a service that a component provides must be given on both the syntactical and the semantical level. Service syntax specifies such things as the formal components interface, methods for access and their signatures etc. Thus the syntactical segment of the interface describes what the component does. Service semantics specify how the component works, under what conditions does it perform the declared functionality, what guarantees does it give about its performance etc. Interfaces defined in standard component technologies today can express functional properties. Most description techniques for interfaces such as interface definition language (IDL) only cover the signature part. Such techniques are not sufficiently good for expressing extra functional properties of components, such as: accuracy, availability, latency, security.

There are generally two kinds of interfaces. Components can export and import interfaces to and from environments that may include other components. An exported interface describes the services provided by a component to the environment, whereas an imported interface specifies the services required by a component from the environment. [1] Various graphical notations are used but the fundamental principles of exported and imported interfaces are the same.

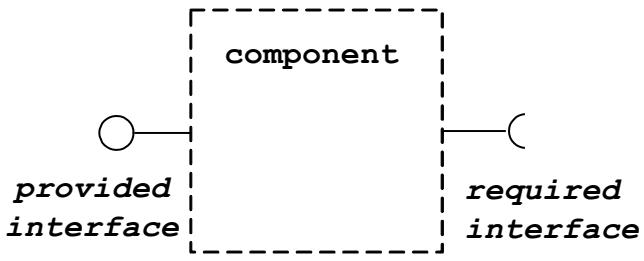


Figure 2.1 Simple component with a provided and a required interface

2.2. Component models and frameworks

Component model is a standard that comprises information about the structure of components interface and ways to implement, document and deploy component that conform to that model. Thus component models are used by software developers that wish to use a certain component model and must ensure interoperability with other models possibly used. Component models are implemented in frameworks, i.e. component platforms. The key contribution of frameworks is that they force components to perform their tasks via mechanisms controlled by the framework, thus enforcing the observance of architectural principles defined in component models.

A framework can be seen as a circuit board in which empty positions are waiting for the insertion of components. The framework (the circuit board) is instantiated by filling in the empty slots. Requirements are specified to indicate to what the components must conform to be able to function as intended in the circuit. The internal details of the specification (the implementation) are still concealed within the component and should remain so. [1] Component frameworks are thus filled with components and instantiated in this way. Examples of ubiquitous component models are Microsoft .NET, JavaBeans, OMG Corba, etc.

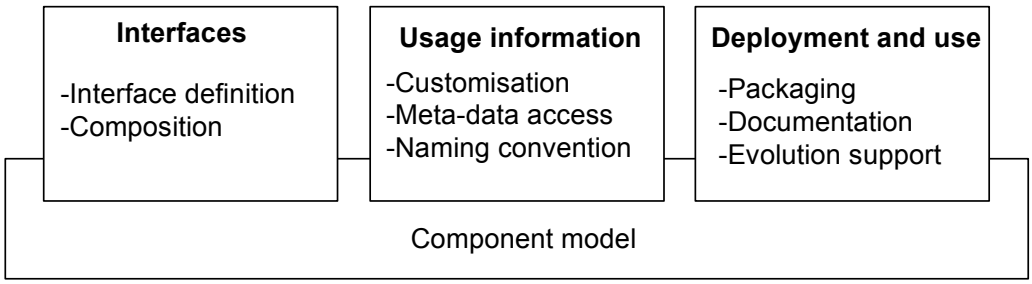


Figure 2.2 Elements of a component model

A component model implementation typically provides two kinds of services [14]:

- *Platform services*, which are the fundamental services that enable communication of components with each other. For example: addressing, interface definition, exception management, component communication, etc.

- *Horizontal services*, which are application independent services which may be reused by many components and systems and thus reduce the development costs and potential incompatibilities. For example: concurrency, security, transaction management, persistence, resource management etc.

2.3. Component composition and the CBSE process

One of the reasons why CBD has not become a major approach in software development is that a fundamental reuse principle hasn't been satisfied which says that the component based development must be an integral part of the entire development process. Thus it is not possible to successfully use traditional development processes for CBD if they have not been

adapted to the details of the CBSE process. Some parts of the process, like gathering user requirements remain the same, however there are certain fundamental differences.

Requirements should initially not be highly specific because very specific requirements significantly reduce the number of potential component models that can be used for development. Therefore, requirements should initially be kept flexible and adaptable to the situation with the available component models. Still, a complete set of requirements is required so that as much reuse possibilities can be identified.
Unlike traditional development, CBSE process contains a component search phase after which the component that best fit the given requirements are selected and adapted. The need for adapting the selected components to the specific details of the project is highly probable.
Development in a CBSE process is then essentially a composition process where the selected and adapted component are composed together and integrated with the infrastructure, i.e. the platform for deployment. Glue code will most likely have to be developed to enable the composition of possibly incompatible components. [14]

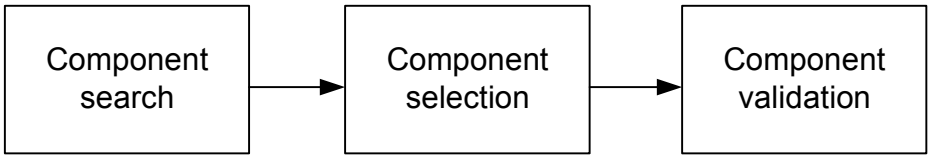


Figure 2.3 Component related elements of the development process

Finally, selected and adapted components are composed into the final system. Assemblies of components are thus formed and deployed on the platform. Ways in which components are assembled and integrated with the platform are defined by the component model. There are various types of composition, such as sequential composition where components are executed in a sequence, hierarchical composition where a component uses services of another component, and additive composition when multiple components form a new component (assembly). Different kinds of problems can occur during component integration, such as: parameter incompatibility, operation incompatibility and operation incompleteness.

3. ProCom – Progress Component Model

3.1. Introduction

As mentioned earlier, goal of this thesis is to analyze various behavior modeling languages and to investigate ways to apply these languages to the ProCom component model. This chapter analyses the component model itself and outlines the elements of the model important for achieving the overall thesis goal.

The ProCom component model is part of the larger project being run at Progress, a national Swedish research centre. Primary interest to Progress is the application of component-based software engineering to the automotive and vehicular domain. The basic idea is that building embedded system from reusable software components results in increased cost-efficiency, increases scalability in handling complexity and integration, as well as a higher quality insurance. The expected outcome of the project is to provide a complete set of theories, engineering methods and tools that can be used for the development of embedded systems.

All this is intended to be utilized for the purposes of achieving the main goal, which is predictable embedded-software development from reusable software components. Other two important goals are interfacing the developed software with the underlying platform, and too apply real-time techniques at all stages of component-based design.

It is worth mentioning that Progress initially declares a general perspective towards the development of embedded vehicular systems, but acknowledges that the component-based approach seems most feasible. Therefore Progress takes a component-based approach to the development of embedded systems for the vehicular domain.

Considering the importance of quality attributes such as security, safety, predictability and resource-efficiency, Progress claims that using a design-time component model is the most promising solution. Design-time component models are composed at design time and the analysis of components and assemblies is also done at design time. This is considerably different from component models such as Microsoft .NET or CORBA. Design-time analysis is essential for achieving predictable behavior and any other kind of early predictions about the system and its properties. In addition to that, the analysis stage is expected to provide all kinds of algorithms and methods for predicting system properties.

In order to achieve predictability, a strong emphasis is put on analysis to provide estimations and guarantees for different system properties. Analysis is present throughout the whole development process and gives results depending on the completeness and accuracy of the models and descriptions. Early, inexact analysis may be performed during design to guide design decisions and provide early estimates. Once the development is completed, analysis may be used to validate that the created components and their composition meet the original requirements. The different analyses planned for Progress include reliability predictions, analysis of functional compliance (e.g. ensuring compatibility of interconnected interfaces), timing analysis (analysis of high-level timing as well as low-level worst-case execution time analysis) and resource usage analysis (e.g. memory, communication bandwidth). [10]

The ProCom component model consists of two layers: the upper layer ProSys and the lower layer ProSave. At the ProSys level, systems are modeled as concurrent subsystems that communicate by passing messages. The lower layer, ProSave, is responsible for the internals of a subsystem which are ultimately defined by the primitive components implemented in code. Unlike ProSys subsystems, ProSave components are passive and their communication is based on a pipes-and-filters paradigm. At both layers, a concept is adopted of a component as a collection of all the information needed to work with that component in a different time during the development process (requirements, documentation, source code, models etc.).

Considering that the goal of this thesis is to analyze the methods for modeling the behavior of a single component, less attention will be given to ProSys, and more to ProSave.

3.2. Overview of ProSys

At the ProSys level, main design element is a subsystem. The entire system is modeled as a set of subsystems that communicate together. It's important to note that a subsystem can be further divided into other subsystems, which means that ProCom is a hierarchical component model.

From a CBD point of view, subsystems can also be conceptually viewed as components. Differences of course exist between components and subsystems in design, implementation and deployment. Different subsystems, i.e. components at the system level, can be deployed to different physical nodes and this is also possible even for elements of a single subsystem.

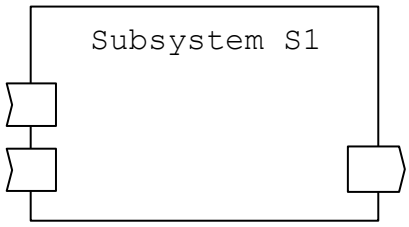


Figure 3.1 ProSys subsystem external view

A ProSys subsystem is specified by typed input and output message ports, as shown in the figure above. The ports describe what kinds of messages the subsystem receives and sends.

Subsystems are active, which means that they can perform activities periodically on their own and not just as a reaction to external stimulus. They can contain reactive parts as well.

Communication between subsystems is modeled with message channels. Channels are used to connect output to input message ports. Message channels thus provide some additional functionality other then being a primitive message carrier, such as the possibility to associate information about the kinds of information being carried, and also to define some data initially required by the system to be available on the channel. Channels support n-to-n communication, i.e. more than one output and input ports may be connected to the channel. Message passing is asynchronous, i.e. sending a message is a non blocking action. The way in which incoming messages are handled is defined by the receiving subsystem.

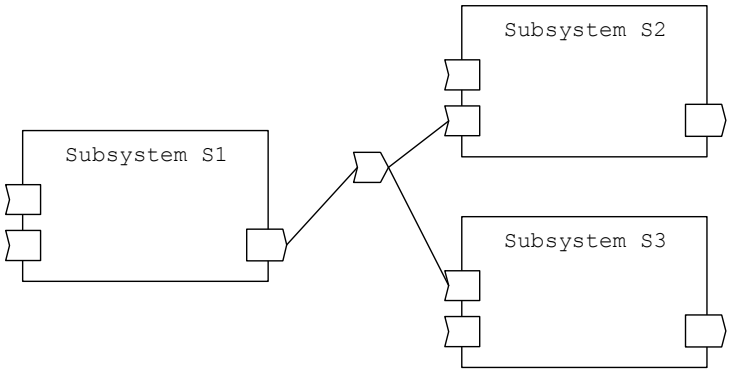


Figure 3.2 ProSys subsystems communicating through a message channel

From the point of view of implementation, there are primitive and composite subsystems. Primitive subsystems are realized internally through ProSave components, by code conforming to the Progress subsystems runtime interface or by wrapping legacy code to make it compatible with the runtime interface.

Composite subsystems internally consist of subsystems and message channels. There are also connections that associate the message channels with message ports of the composite subsystem or the subsystems inside. This allows an input port, acting as a message consumer outside the component, to act as a message producer internally. Oppositely, an output port consumes messages on the inside and acts as a message producer from the outside.

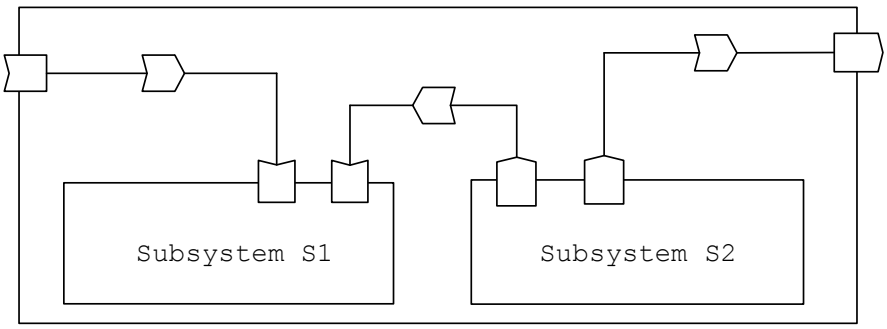


Figure 3.3 ProCom composite subsystem

3.3. Overview of ProSave

ProSave is a design language primarily intended for developing primitive ProSys subsystem with complex control functionally. Naturally, it is component based and a component is the fundamental building block in ProSave. A ProSave subsystem is thus constructed from interconnected components. Component can also be hierarchically structured.

ProSave components are completely passive, which means that they never initiate any activities on their own. Components don't have their own execution threads. They must therefore be activated by some external entity through some stimulus. After they have been activated (stimulated), they perform their functionality and return to the passive state.

It is important to note that ProSave components are design-time entities that do not exist as individual units in the final system that is being executed on some physical machine. During the deployment process, they are transformed into operating system tasks, which greatly improves efficiency of the final system, and this is of great importance in the target domain of ProCom.

ProSave architectural style is based on data/control flow that explicitly separates the flow of data from the flow of control.

In the following two sections, structure and the behavior of ProSave components are analyzed. The structure of a component is only its externally visible interface, whereas the internal structure is actually the components behavior.

3.3.1. Structure of ProSave components

A ProSave component is the fundamental building block when constructing subsystem at the ProSave level. It is intended to contain small and non-distributed functionality. Component external structure consists of two major elements: ports and attributes. Component ports provide the way to access components functionality, and component attributes provide the way to access information about the component.

A ProSave component is a black-box from the outside, i.e. its internal implementation is not visible or directly accessible. This black-box view is considered reasonable, however some more complex analysis procedures may require a white-box view of a component, i.e. access to its internals. For example synthesis will definitely require a white-box view of a component considering that components are ultimately indistinguishable in the final system.

The separation of the data flow from the control flow is done by having two separate types of ports: data ports and trigger ports. The data flow is carried through the data ports and the control flow is carried through the trigger ports. From the direction point of view, both trigger and data ports can be of input and output type. Trigger ports are depicted as triangles and data ports as squares.

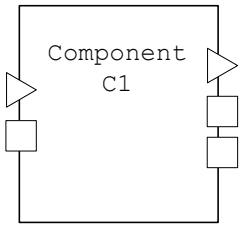


Figure 3.4 ProSave component with two input ports (left) and three output ports (right)

Component behavior is modeled through the concept of services. When multiple services are provided by a single component, it is necessary to enable separate external access to them, and it is done by forming group of ports.

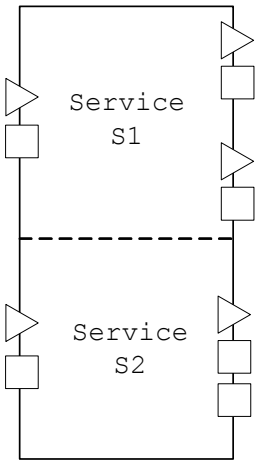


Figure 3.5 Port groups on a ProCom component with multiple services

3.3.2. Behavior of ProSave components

Internally, the functionality of a component can be implemented in code (primitive component), or by interconnected subcomponents (composite component), but the distinction is not visible from the outside. Only primitive components are considered here, since composite components are ultimately also built from primitive components.

As described earlier, a ProSave component is passive, i.e. it never initiates any activities on its own and it does not have its own execution thread. Component can only react to some external stimulus, after which it will perform its functionality and produce some response on its output ports. Taking into consideration the data/trigger paradigm in ProSave, when the input trigger port is activated, it reads the current value at the input data port and starts processing this value according to the implemented functionality. When the calculation has finished, the result is produced at the output data ports on the components right side, and control is forwarded/returned via the output trigger port.

The concept of service was introduced in the previous section. Component will always provide at least one service, however it can provide more than one. This means that different services within a single component can produce parts of the result at different points in time. Each service therefore corresponds to some particular functionality that a component provides. Services are triggered independently and may run concurrently. Following elements constitute a single component service:

- input port group, that contains a single trigger port for service activation and a set of data ports for providing required data to the service

- set of output port groups, that contains a single trigger port for forwarding the control and a set of data ports for providing the result of services functionality.

The port to port-group relation and the port-group to service relation are clear from the figure above.

During its lifetime, a service state is changed from active to inactive and vice versa. Initially, all services of a component are in an inactive state. In this state it is possible for them to receive data and trigger signals on the input ports, but no computation is performed internally. When an input trigger port is activated, i.e. when a control signal comes to the component, all the data ports in the group are read in one atomic operation. When the data has been gathered the service switches into the active state and it performs its computation and produces output at the output group. The data and triggering of an output group are always produced in a single atomic step. When in the active state, a service can not be triggered again, i.e. attempts of reactivation on the trigger ports are ignored. This must be enforced by a design-time checker, and not at runtime to ensure high efficiency.

The functionality of each service is implemented by a non suspending C function. The component also has an init function which is called at system startup to initialize the internal state. A primitive component specifies a header C file and a source C file, where the init function and the service entry functions are declared and defined. The header file also declares the structs used for input and output to the services. By default, the naming of entry functions and argument structs is based on the names of services and ports, but explicit name mappings can also be supplied. [10]

During synthesis, the design-time components are transformed into runtime entities, such as operating system tasks. It is the responsibility of synthesis to ensure that the behavior of the runtime system is consistent with what is specified by the execution semantics and the ProSave design. For example, although the semantics view data transfer on different levels of nesting as separate activities, the final system may realize communication between two primitive components on different levels by a single write to a shared variable, ignoring the intermediate steps of activating input and output port groups, as long as the overall behavior is consistent with the execution semantics. [10]

Restrictions on data/control delivery are not complex. One rule is that trigger signals should not arrive to trigger ports before all data has arrived at data ports. Furthermore, when data reaches a port, it immediately overwrites the current value of that port. Also as mentioned earlier, triggering of a service in a active state is ignored. If the service is in the inactive state, then the values of the data ports of the triggered port group are atomically read, the state changes to active and the component performs its functionality.

4. CCL – Construction and composition language

4.1. PECT perspective

CCL is part of the PACC (Predictable Assembly from Certifiable Components) initiative from the Software Engineering Institute at Carnegie-Mellon University. The PACC initiative has taken the prediction approach to CBD. It is the PACC position that predicting the behavior of components and assemblies is the only feasible future for CBD. It is therefore in the PACC objective to investigate and develop supporting technologies for making such predictions in a systematic and standardized way.

Considering that predictability is a fundamental issue in PACC it is necessary to define what a predictable assembly really is. Assembly of components is defined to be predictable if its run-time behavior can be exactly determined from the properties of its components and their interaction mechanisms. This is a simplified definition of predictability and it is discussed in more detail later.

Another fundamental issue in PACC is the certifiability of components. A component is said to be certifiable if its properties can be measured or verified by third-parties.

It is important to note that PECT is one possible approach to PACC. The fundamental assumptions of PACC are always the same: achieving assembly predictability from certifiable components. PECT offers one solution to this problem through the use of reasoning frameworks. PECT is, in short, a platform for predicting assembly behavior. It is therefore required from the PECT to support (1) the analysis of component and assembly properties and (2) the prediction of component and assembly properties. A PECT is generally made up of two parts: an analysis technology and a component technology.

It's important to emphasize that PECT is only a concept (a blueprint) that must be applied to some concrete component technology (component model). In order to „transform“ a existing component technology into a PECT, the constructive segment must be adapted, and an analytical segment must be defined and added. Through this process, a new component technology is created (PECT).

As mentioned already, a PECT has two main parts: a construction framework and one or more reasoning frameworks. Reasoning frameworks are connected to the construction framework via interpretations.

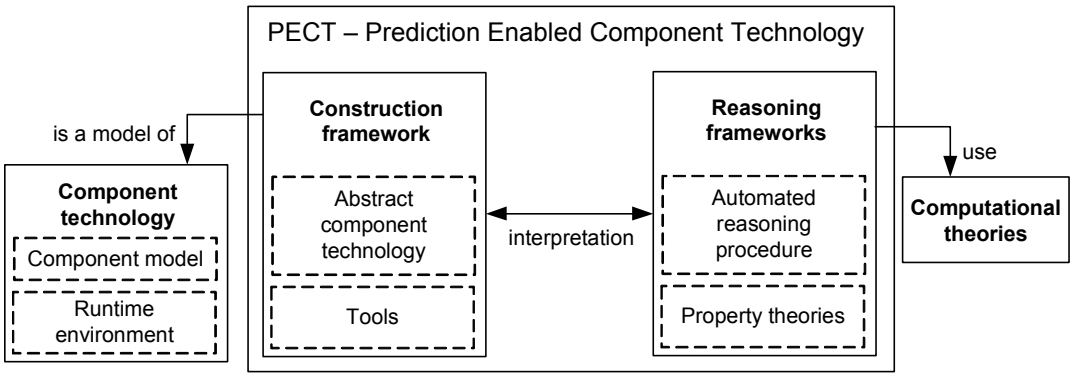


Figure 4.1 Logical structure of Prediction-Enabled Component Technology

Each concrete component technology is „transformed“ into a PECT through the concept of the construction framework. The construction framework essentially consists of two elements: abstract component technology (ACT) and a construction language. It is the purpose of the construction language to describe a concrete component technology into a ACT. Therefore the ACT is a description of a particular component technology written in the construction language.

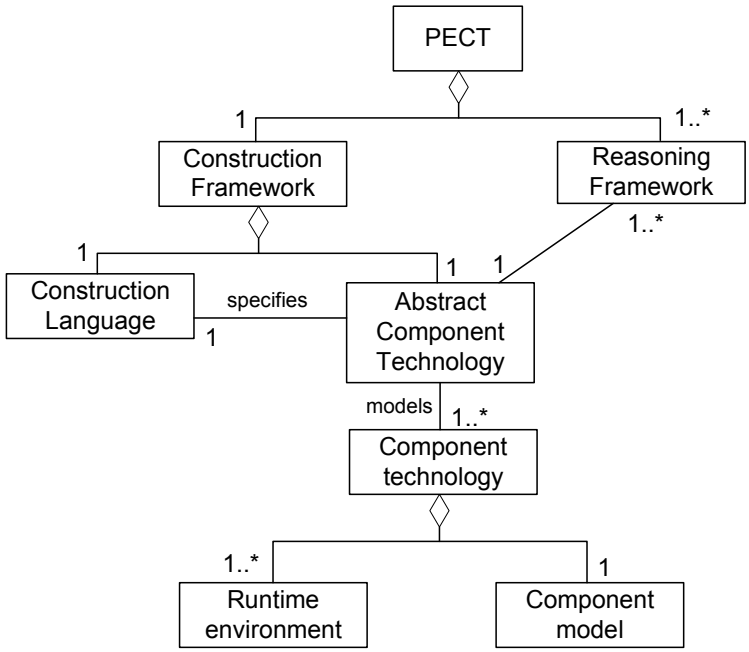


Figure 4.2 UML diagram of PECT concepts

In this case the construction language is the Construction and Composition Language (CCL). The purpose of using a single language is to enable the use of existing PECT tools with any component technology that has been transformed into a PECT.
A second purpose of the language is to enable the description of assemblies in the given ACT.

Therefore the construction language (CCL) enables, among other things, the description of:

(1) Assembly structure (component composition, etc.)

(2) Component behavior, interaction mechanisms (from the model), services (from the runtime)

(3) Properties required by different reasoning frameworks (these properties can be attached to different elements in the specification)

The majority approach to predicting the behavior of software systems today is software testing. Testing must be done very rigorously and the test results are of value only for the system being tested. Therefore the fundamental assumption is that by watching and analyzing the past, we can predict the future.

The PACC project takes a radically different approach and aims to enable prediction by creating analytical theories for entire classes of systems. As soon as a system satisfies the assumptions of a particular theory, it is predictable within that theory. The result of this is a specific view of the „world of all assemblies“ where we only recognize assemblies that are well-formed with respect to a analytical theory and thus that are predictable. All other assemblies are not predictable. The main goal then is to only build assemblies whose behavior can be predicted and we don't (and can't) predict the behavior of arbitrary non-valid assemblies. Therefore analytical theories impose certain design constraints.

The focus in the PACC project is on developing analytical theories for system properties of significant business value such as: reliability, safety, security etc.

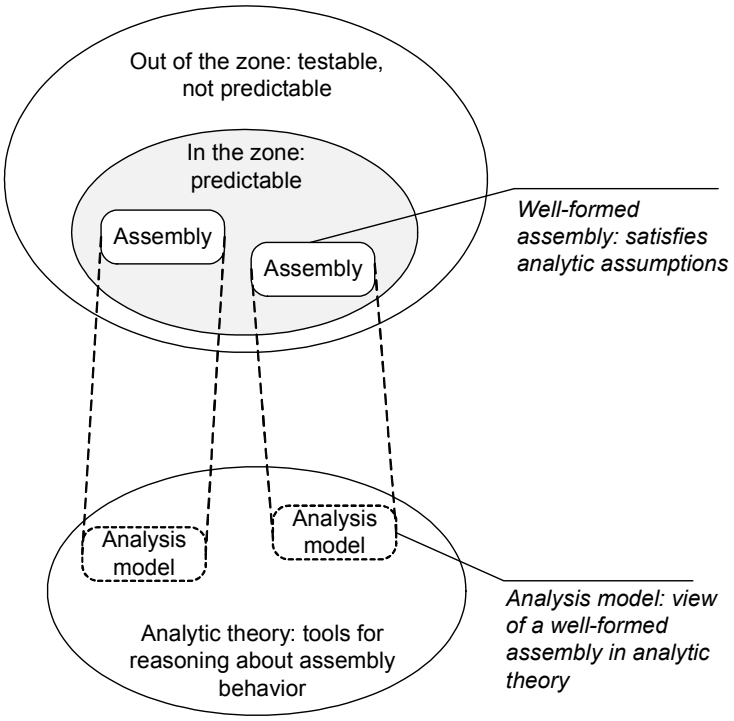


Figure 4.3 Relation of the predictability zone and the analytic theory

The bases of an analytical theory are the assumptions that the theory makes and these assumptions influence the required knowledge about the component and confidence in it. For example a safety theory might require that a component specification contains a state machine of component operations, etc. So its obvious that with regard to analytical theories

components cannot be idealistic black boxes because the theories require insight into component implementation and internal behavior.

This information required by the theories is acquired from the component through a analytic interface. These interfaces provide analysis-specific views of a component. A confidence in these interfaces is also essential in achieving overall prediction confidence. Therefore the PACC project has established such a foundation for building trusted and certified components.

It can be said that two fundamental premises for achieving predictable assemblies are achieving predictability by construction (building assemblies whose behavior can be predicted) and trusting the predictions with enough confidence. Confidence is achieved through rigorous empirical and formal validation because it is important to establish quantifiable trust.

4.2. Construction framework

The construction framework is one of the two essential elements of a PECT. The other essential element are the reasoning frameworks which are connected to the construction framework via interpretations. Purpose of the construction framework is to support all of the usual construction activities done in component-based development.

Key feature of the construction framework is that it supports CBD construction activities in a technology-independent way. This is the purpose of ACT (Abstract Component Technology) which is an important part of the construction framework. The ACT is fundamentally a proxy for concrete component technologies. The ACT provides the language and notations for specifying component, assemblies, runtime environments etc. but in a technology-independent way. Along with the ACT, the construction framework comprises tools that provide support for construction activities.

Following is a description of essential CBD concepts and the way they are utilized in an ACT.

As in CBD in general, a component is the fundamental building block of assembly. In contrast of typical CBD component definition as a black-box during usage, in the PECT view it is clear that much of components inner structure must remain exposed for the purpose of prediction and certification. In the context of a PECT, a component is considered an implementation that provides interfaces for third-party deployment through binding labels (pins) and can be independently deployed. In PECT, components final form is an executable binary, and not source code. A PECT component can be considered a unit of independent deployment if it has all of its dependencies on external resources specified and if it can be substituted by, or substitute of, another component. [3]

Components interfaces are realized through incoming sink pins, and outgoing source pins. Sink pins are responsible for incoming events to a component, whereas source pins are responsible for events outgoing from a component. The sink/source pin paradigm is equivalent to the provided/required interface paradigm present in other literature.

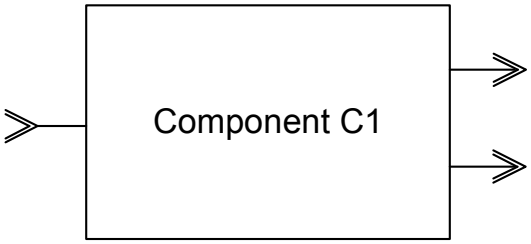


Figure 4.4 Simple CCL component

Component behavior is specified in the form of reactions. Reactions essentially describe the connection between components sink pink and its source pins. This means that reactions define ways in which the component will react to external stimulus on its input pins. The fundamental reaction that a component can perform is to emit a event as a response to a incoming stimulus.

The minimal description of a reaction contains only information about sink and source pins involved in the reaction, however a more complex mechanism for specifying reactions is required. PECTs have primarily used CSP process algebra for describing reactions. The main drawback with CSP is its complexity. Recently, CSP has been replaced with a action language based on UML state charts. The state chart version is described in the CCL section.

It is important to note that reactions can be described through formal implementations in languages such as Java, or they can be described in an abstract way with languages such as CSP. Only requirement is that the reactions specification is parsable.

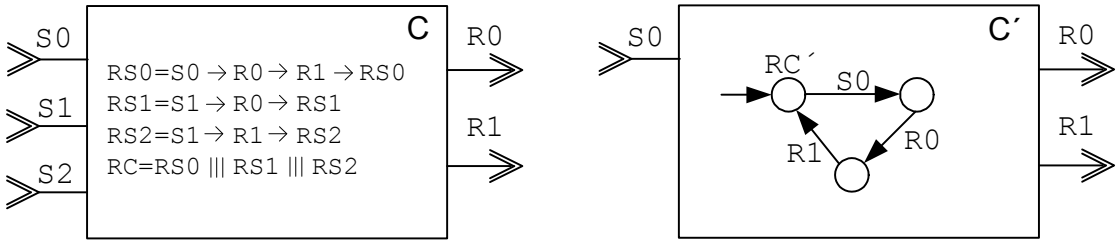


Figure 4.5 Reaction specification in CSP (left) and an equivalent transition system (right)

Except for internal behavior of components, there is also external behavior among components, i.e. their interactions inside an assembly. Interactions describe composite behavior of multiple components, or more precisely, of multiple reactions. It is naturally required that components are interconnected before they can interact with each other. Components are therefore considered connected/composed if their pins are connected together. The existence of this connection between pins means it is possible for a interaction to occur but it is not obligatory. The behavior of a composition of components can be deduced from the behavior of its constituting reactions.

The PECT expects that the utilized concrete component technology has some form of a runtime environment. The term runtime environment is in other literature also known as: framework, container, platform etc. No matter what the name is, common and fundamental functionality is important. The environment [3]:

(1) provides services to components and assemblies (transaction, security, etc.).
(2) manages resources (thread pools, database connections)
(3) manages component life cycles.

The environment can therefore be considered as a special kind of component with which other component/assemblies/environments may interact. It must also provide a concrete implementation of the interaction mechanisms (communication protocols) for the components to use. The environment must also provide support to assumptions required by reasoning frameworks. In this respect, the environment can be considered a component-aware virtual machine [3]. Structurally, the difference between components and assemblies/environments is that components cannot have internal structure whereas assemblies and environments can and must.

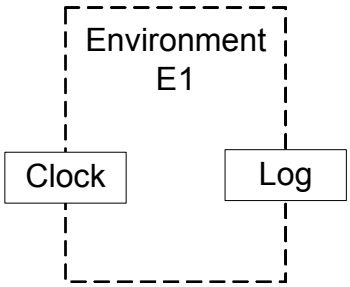


Figure 4.6 Simple CCL environment with two services

Generally there are synchronous and asynchronous modes of communication, and pins can further specialize this modes. Synchronous communication mode can be specialized by pins denoted in the construction framework language as sink mutex which enforce mutual exclusion on reaction within a component or by pins denoted as sink reenter which permit concurrent behavior of reactions. Asynchronous communication mode can be specialized by pins denoted as unicast or multicast which is self-explanatory. Synchronous pins can be visually distinguished from asynchronous pins because they have single-lined pin heads, whereas asynchronous pins have double-lined pin heads.

In principle, asynchronous communication is based on exchanging messages between parties communicating together. Synchronous communication, on the other hand, involves some kind of a procedure call, i.e. it requires some kind of a established communication channel between the communicating parties.

Summary of interactions provided by an environment:

- Asynchronous (event-based) interaction through an unbounded priority queue. Priorities are taken from the priorities of reactions that emit them.

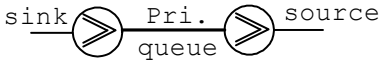


Figure 4.7 Unbounded asynchronous interaction in CCL

- Asynchronous (event-based) interaction through an bounded FIFO queue of length n.

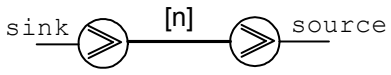


Figure 4.8 Bounded asynchronous interaction in CCL

- Synchronous (call-return) interaction through a semaphore acquired by the calling reaction, i.e. the reaction is in a protected critical section.

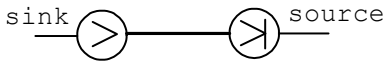


Figure 4.9 Synchronous interaction in CCL

Components by themselves of course have no purpose and they must be composed into assemblies. Assembly is therefore a set of components that can interact. Assemblies themselves have no behavior of their own, i.e. their behavior is indirectly the behavior of their components. Assemblies can be contained by other assemblies, thus forming a structural hierarchy. An assembly provides constructive closure to a component which means that all components interactions are confined only to its immediate assembly. Since interactions among components are provided by the environment, each assembly is associated with exactly one environment.

Assemblies are subject to the same type/instance paradigm as well as components and environments. For the purpose of achieving assembly hierarchy, i.e. having assemblies inside assemblies, an assembly can expose a particular pin that belongs to one of its contained components.

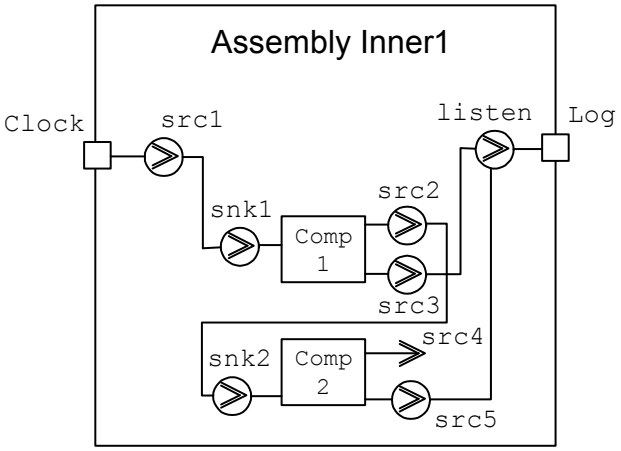


Figure 4.10 Simple CCL assembly Inner1 in environment E with log and clock services

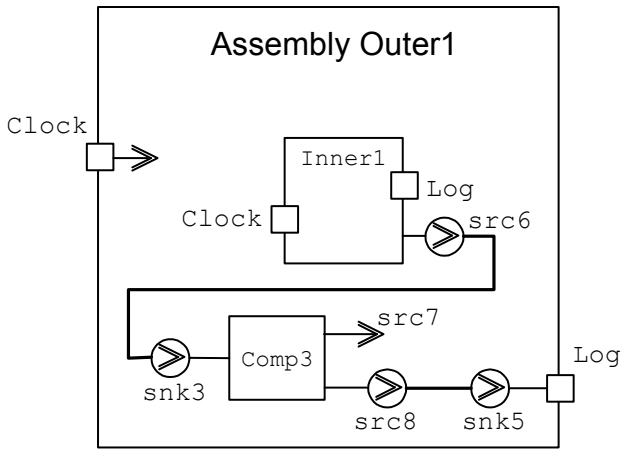


Figure 4.11 Assembly hierarchy through exposed pins

4.3. Reasoning frameworks

The purpose of the reasoning framework is to analyze and predict the behavior of assemblies based on the description of properties of components and assemblies. Reasoning frameworks contain concepts and theories required to analyze and predict assembly qualities. Naturally, different reasoning frameworks are required to predict different runtime qualities. They use different component properties as inputs and give different requirements on what is and what is not an analyzable design.

The reasoning framework is essentially a way PECT packages the complexities of model checking technologies combined with component technologies. This allows the developer to predict the behavior of component-based systems without having expertise in analysis technology.

The interpretation is the link between the ACT (only 1 in a PECT) and each reasoning framework (at least one in a PECT). The purpose of the interpretation is to relate concepts from both frameworks. More formally, the interpretation is used during the translation of the construction language assembly specification into a reasoning framework understandable language. Therefore, reasoning frameworks are applied to assembly specifications by means of formal interpretations that generate analysis models from CCL specifications. Most interpretations use general CCL information, but specific analysis information is provided through CCL annotations.

The reasoning framework used in PACC is Comfort. Comfort is a model checking reasoning framework that provides a way to incorporate model checking techniques into a PECT.

Fundamentally, formal verification implies having a mathematical model of a system and describing system properties in a formal language. If the system behaves according to its specifications, then it is said that the model satisfies the specification.
What model checking brings is automation, thus model checking is a completely automated form of formal verification that checks if a system satisfies a constraint by exhaustively searching through all possible states of the system. This exhaustive searching eliminates classical testing coverage problems. Model checking verifies finite-state concurrent systems.

This limitation allows complete automation. Also, model checking will always terminate with a yes or no answer, i.e. the model satisfies or does not satisfy the constraints.

It is important to note that the restriction to finite-state systems is not a disadvantage because it is still applicable to a lot of important classes of systems, such as hardware controllers, communication protocols, etc. Non-finite software may be verified if variable values are assumed over finite domains. This is also not a restriction because many important classes of systems, such as message queues can be restricted from infinite to finite queue lengths.

4.4. Component certification

Achieving assembly predictability is a fundamental issue in PACC. A crucial condition for this is trusting the components that make up the assembly. PACC acknowledges that components are possibly subjects to third-party composition, which is reason enough to have a sound mechanism for providing trust in components. Since components are now days mostly shipped as binary code ready for execution it is important to create certified binaries, i.e. binaries that are trustworthy. But for a consumer to believe that a component is trustworthy, he must be able to verify this in a formal way.

A core idea used in PACC is proof-carrying-code (PCC) concept. Considering that a consumer will always request that a component satisfies a certain policy, the idea of PCC is to create a proof that the component satisfies this policy, embed the proof into the component binary and then ship it as a single unit. A consumer can then verify the validity of the proof on his own.

Policies most used today are safety policies, while proofs require considerable resources because of their size. PCC expands the policy set to include both safety and liveness policies. A linear temporal logic called SE-LTL is used to specify these policies. SE-LTL was also developed as a part of the PACC project. PCC uses automated model checking techniques for generating invariants and ranking functions required for proof construction. PCC also uses state-of-the-art SAT (Boolean satisfiability) technology to reduce the size of proofs to minimum.

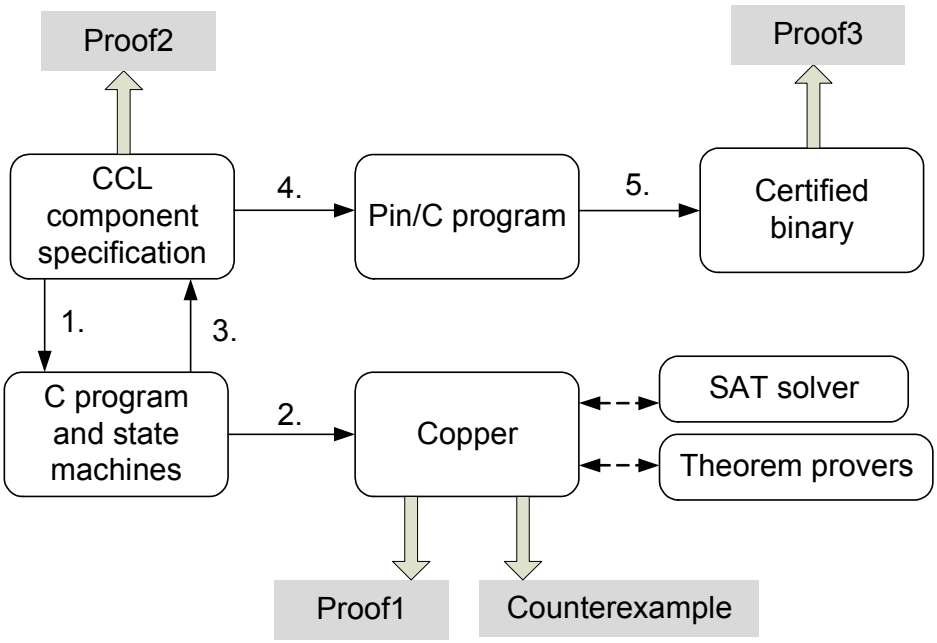


Figure 4.12 PECT certification procedure

Figure above describes the certification procedure. Certification procedure begins with a CCL specification of a component assembly. The specification will contain a structural and behavioral description of the assembly as well as the safety and liveness policies that need to be certified. Required policies are specified in SE-LTL temporal logic.

The specification is then transformed (step 1.) into a form understandable by a model checker. This form is made up of C code and finite state machine specifications.

The model is transferred (step 2.) to the software model checker Copper. During its work Copper uses functionality of external SAT solvers and theorem provers to produce results. One possible outcome from Copper is a proof (Proof1) that the given model in fact satisfies given policies. The other possible outcome is a counterexample demonstrating why the model doesn't satisfy the policies. What Proof1 really means is that the (C+FSM) model of the given CCL specification satisfies the given policies.

This must also be proven for the CCL specification itself which will establish the correctness of the interpretation from the specification to the model. Therefore Proof2 is generated which proves (step 3.) that the CCL specification was correctly reverse-interpreted from the model.

The CCL specification is transformed into a Pin/C program runnable in a Pin runtime environment. This transformation works on the CCL specification, but also on the proof (Proof3) that the specification is correct, thus generating proof that the Pin/C program is also correct and satisfies given policies.

Finally, using a compiler, the program is transformed into binary code (step 5.)

4.5. Pin component technology

The Pin component technology is the only concrete technology that has so far been utilized for purposes of a PECT. As every component technology, Pin is comprised of a component model and a runtime environment. The component model defines the logical level of component and application structure as well as general interaction rules. The runtime environment, on the other hand, implements these rules and provides basic services, such as resource management, communication etc.

Pin has been completely developed at the SEI institute as a separate component technology, but has been developed for use in a PECT.

The target class of systems are embedded, safety-critical and time-critical systems. This means these systems are very small in implementation. It also means the implementation is very transparent because a lot of hiding would be too costly, unnecessary and thus counterproductive for this class of systems. Pin has been successfully implemented and used in real applications.

It is important to note that generally PECT is not formally restricted to the usage of Pin. Since PECT is a concept, i.e. a blueprint, it can be applied to any concrete component technology. However up to date, only Pin has been successfully used for a PECT implementation.

While designing Pin, some overall design objectives were kept in mind. (1) A simple programming model was required, as well as a very simple execution model that supported the UML state chart semantics. This requirement is important from both the usability and automation viewpoint, since a simple model simplifies the translation of code. Also changes to the Pin implementation are less likely to reflect on code generators. Also UML, as a widely spread language, enables the specification of component behavior as well as a basis for formal development, in a standard language. (2) A mechanism was needed to enable enforcement of external design and implementation constraints. (3) Only most basic features required for a PECT were introduced.

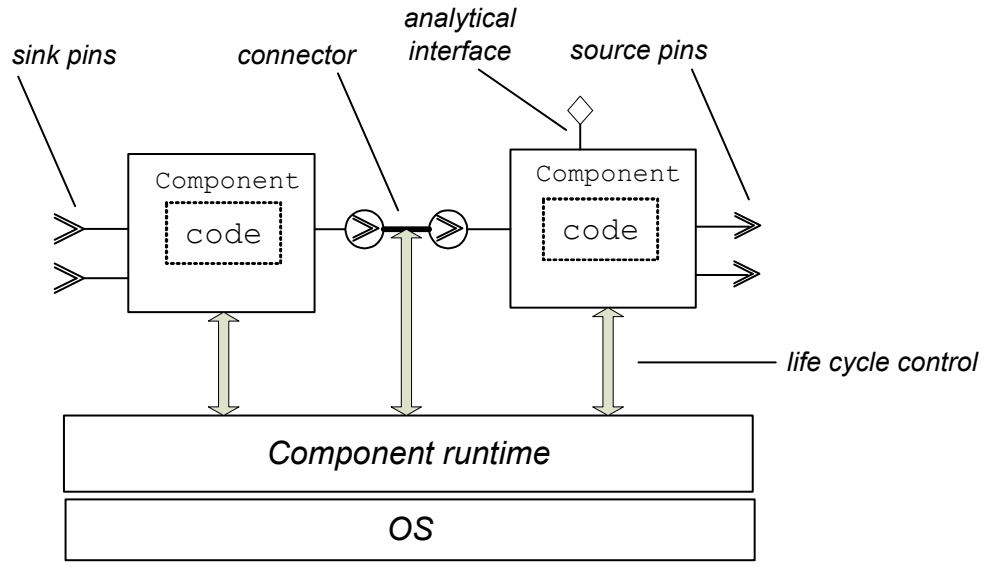


Figure 4.13 Pin logical structure

4.6. PACC Starter Kit

The PACC starter kit is a complete software solution with all the technologies developed in the PACC project combined together to provide a usable tool. The starter kit is based on Eclipse.

Main functionality provided by the starter kit:

- specification of components and assemblies of components in CCL.
- prediction of component and assembly behavior through behavior analysis
- generating binary code that implements the specifications

For the purpose of component and assembly specification, the Kit provides a CCL parser and the CCL compiler to generate implementations of the specifications in the target Pin component technology. The Kit thus contains the Pin component technology, i.e. the component model API and the runtime environment.

From the analysis and prediction perspective, the Kit contains a performance analysis reasoning framework that can be used to analyze worst and average performance of component and assemblies specified in CCL. A generalized rate monotonic scheduling theory is used as a supporting analytical theory. The Kit also contains a behavior analysis reasoning framework for behavior verification with model checking as the supporting analysis technology. A security analysis reasoning framework is also included for finding buffer overflows in C programs also with model checking as the supporting analytical technology. Finally, a memory analysis reasoning framework is included for predicting minimum page file sizes used by assemblies.

4.7. CCL – Construction and composition language

PACC initiative is based on using PECT's (Prediction Enabled Component Technologies) which include a component technology and one or more reasoning frameworks. The purpose of PECT is to enable the prediction of component and assembly behavior with respect to those frameworks. The prediction process is based on certifiable properties of components and their specifications. The language used for writing such specs is CCL. So CCL specifications represent everything a reasoning framework needs to know in order to predict component and assembly behavior. There are also a number of automated tasks in the entire prediction process that can be automated with the help of CCL.

Information that CCL specifications provide can be divided into three general groups:

- structural information
- behavioral information
- analysis specific information.

Only structural and behavioral specifications will be further analyzed as they are most significant in defining a component.

Specifically a concrete CCL specification contains:

- component annotations – used for supplying additional information to the Pin code generator (optional)
- variable declarations – standard programming language variables (optional)
- structural specification – description of components externals (pins)
- behavioral specification – description of components internal behavior (reactions)
- verbatim code – CCL mechanism that enables the use of arbitrary C code as part of components behavior

CCL is formally defined through a language grammar. The starter kit provides an environment that supports the creation, validation and further usage of a CCL specification. The grammar is textual therefore CCL specifications are created and used only in pure textual form. Graphical notation may be used informally when specifying component structure and behavior but it is currently not supported by the development environment.

Structural features of CCL are later introduced in graphical notation, and then in the formal text notation as they are defined in the grammar. The graphical notation used corresponds to the way components and assemblies are represented in the Pin component technology. Usage of informal Pin notation makes sense since Pin concepts are mirrored in the language. It is also important to mention that CCL is therefore an IDL to the Pin component technology.

Behavioral specifications of components may be informally given in standard UML state-chart notation given that the action language is based on the state-chart language. However a CCL specific syntax is used for the textual representation of UML state-charts. CCL is therefore also a surface syntax for UML state-charts.

Part of the language responsible for specifying component behavior is called the action language. The CCL action language is C-like in syntax but is much more restrictive than C. An example of a restriction is the absence of pointers in the language. CCL follows the general C syntax for variable declaration: the type name and variable name are followed by an optional initializing expression. The initializing assignment occurs exactly once, when the component is instantiated. CCL supports the string, boolean, enum and several int and float types. Multi-dimensional array types can also be defined using C typedef syntax. Arrays in CCL are much more restrictive than C arrays, for example, the bounds of all array dimensions must be statically defined.

4.7.1. Structural elements of specifications

Primary focus here is on the specification of a single component. In CCL, as well as CBD in general, a component is the fundamental structural element of an assembly. The PACC project recognizes the concept of a component in the context of component types and component instances. In CCL the concept is formalized through the component keyword used in formal specifications of a CCL component type.

The structural part of the component specification specifies how the component interacts with its environment. In CCL the only way a component interacts with its environment is through its pins. There are no other ways of communication with the component. The structural specification defines for example the type of communication with the environment, type of data exchanged with the environment etc.

There are two types of pins: sink pins and source pins. The component receives stimulus for communication only through its sink pins and it initiates communication with the environment only through its source pins. Pins are formalized in CCL through the use of sink and source keywords.

Figure below shows the graphical syntax used in CCL for describing the components structural information.

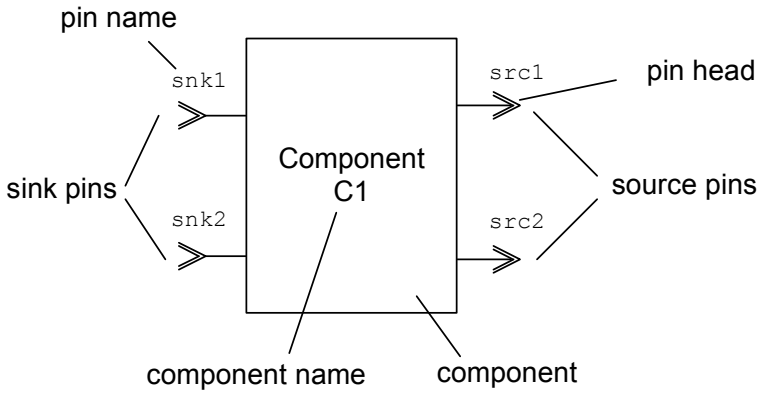


Figure 4.14 CCL component structural information

By convention, sink pins are placed on the left side of the component, and the source pins are placed on the right side of the component, which according to the CCL specification, „allows, at the cost of some cultural bias, to read the component from left to right".

Generally there are synchronous and asynchronous modes of communication, and pins further specialize these modes. Type of communication supported on a pin is defined in the component specification using the synch and asynch keywords.

Synchronous communication mode can be specialized by pins denoted as sink mutex which enforce mutual exclusion on reaction within a component or by sink reenter which permit concurrent behavior of reactions. Asynchronous communication mode can be specialized by pins denoted as unicast or multicast which is self-explanatory. Synchronous pins can be visually distinguished from asynchronous pins because they have single-lined pin heads, whereas asynchronous pins have double-lined pin heads.

In principle, asynchronous communication in CCL is based on exchanging messages between parties communicating together. Synchronous communication, on the other hand, involves some kind of a procedure call, i.e. it requires some kind of an established communication channel between the communicating parties. However, CCLs specification notes that it is not always implemented in this way.

Type of data that can be transferred through component pins is defined by the pin data interface. The data interface of a single pin is a set of simple data types that can be transferred through the pin. CCL supports transfer of very common data types such as: int, short, byte, bool, float, double, string, void or custom types.

In the figure below is an example of a simple component and its basic CCL structural specification. Component has a single sink pin snk1, and source pins src1 and src2. All pins

are intended to be used for asynchronous communication. None of the pins have declared arguments which mean this component will not exchange any data with the environment.

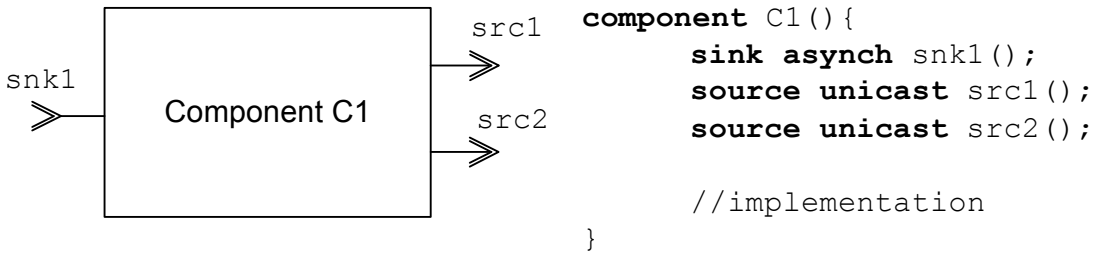


Figure 4.15 CCL component structural specification

The component runtime environment is also modeled in CCL and it also conforms to the type/instance paradigm. The keyword environment is used for defining a environment type specification. Environments generally have two important roles which are implemented in CCL, they provide services and interaction mechanisms. Detailed CCL specifications of environments are expected to be delivered by PECT developers, since environments are expected to be reused for multiple systems.

Services are defined in the environment through the keyword service. They are basically simple components maintained and provided by the environment to its containing assemblies and indirectly components. The reason why the keyword component was not used for the specification of services is that differences in well-formedness rules may exist between environment components and application components. A very specific difference between services and components is that components are purely reactive, i.e. they cannot begin activity on their own, whereas services can, in fact it is expected of them. Another fundamental difference is that services represent environment supplied code that can communicate with the world outside of the assembly (such as device drivers), while all component communication is explicitly represented with its pins. This is why service instances are shown on the border of the assembly instance, as will be seen later.

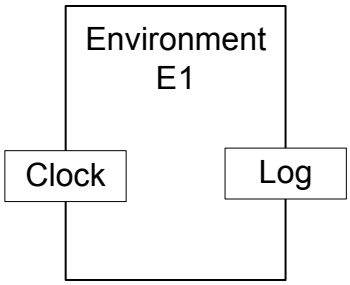
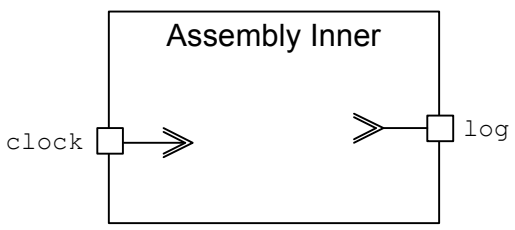


Figure 4.16 CCL services in an environment

Assemblies are essentially groupings of components that work together to provide some functionality, i.e. functionality of the assembly. In CCL assemblies also follow the type/instance paradigm like component, environments and services. Assemblies are defined using the keyword assembly which introduces a new assembly type. In spite of that, the difference between assembly type and assembly instance is not always as clear as the one between, for example, component types and component instances.

Assemblies can exist only inside a environment which provides the services and interaction mechanisms they require. Interaction mechanisms between components inside an assembly rely on the mechanisms provided by the environment.

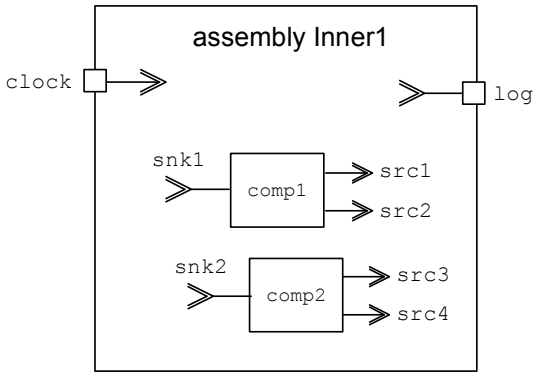
Definition of an assembly in CCL must therefore be explicitly related to a specific environment. This is done through parameterization as shown in figure 1.13. When an assembly is defined in an environment with existing services, the quantity of service instances is unknown in advance. Therefore the assembly specification can make an assumption about this using the keyword assume.




```
assembly Inner()(E){
        assume{
                E:Clock clock();
                E:Log log();
        }
        //implementation
}
```

Figure 4.17 CCL assembly specification

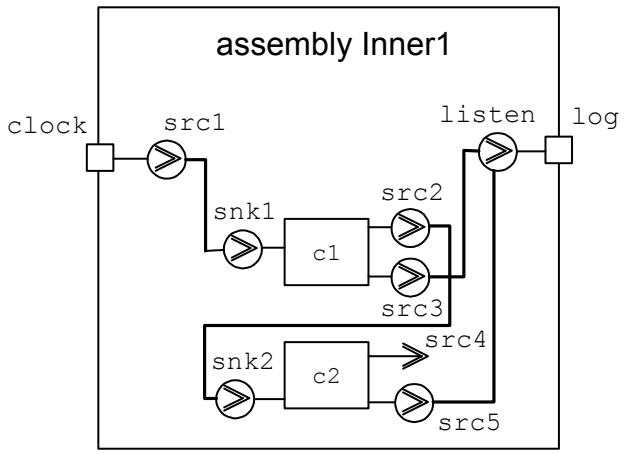
Components are added to the assembly by instantiation. In order to enable component interaction they must be connected. In CCL components are connected using the ~> operator. It connects the left operand (source pin) to the right operand (sink pin). The language specification defines several rules that connections must conform to, for example: source and sink pins must have compatible interaction types (unicast source is not compatible with mutex sink but is compatible with asynch sink), each consume data parameter must correspond to a produce parameter and their types must also be compatible,




```
assembly Inner()(E){
        assume{
                E:Clock clock();
                E:Log log();
        }
        Comp comp1(),comp2();//instances
        ...
}
```

Figure 4.18 CCL component instantiation within an assembly

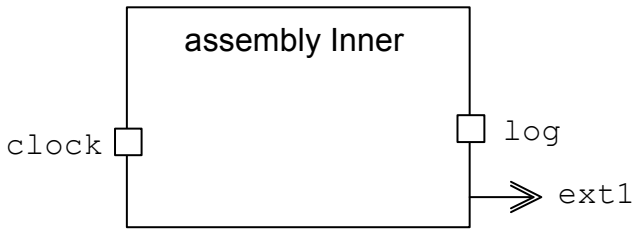
As mentioned earlier, CCL specifications also contain analysis specific information. This information concerns limitations imposed by the reasoning framework. A reasoning framework may specify well-formedness rules which might have to be imposed through additional properties in the CCL specification. CCL uses the annotation mechanism for this purpose. Example of such specific properties, for example thread priority specification is shown in figure 1.15.




```
assembly Inner()(E){
        assume{
                E:Clock clock();
                E:Log log();
        }
        Comp c1(),c2();//instances
        // properties
        annotate c1 {
          "passIt:priority", const
                int priority = 15}
        annotate c2 {
          "passIt:priority", const
                int priority = 10}
        //interactions
        clock:src1 ~> c1:snk1;
        c1:src2 ~> c2:snk2;
        c1:src3 ~> log:listen;
        c2:src5 ~> log:listen;
        ...
}
```

Figure 4.19 Example of providing analysis specific information through CCL

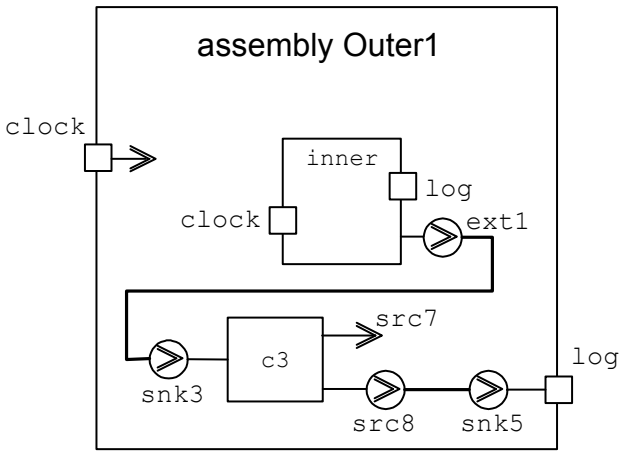
Considering that assemblies by them selves have no structure except from their components, and thus have no interface on their own, it follows that there must be a way to connect assemblies together. Connecting them together actually means connecting their inner components together. Since their components are not directly visible or accessible from the outside, CCL provides a mechanism for exposing certain component pins from inside the assembly. Other assemblies with their exposed pins can then connect to this assembly through its exposed pins. Not surprisingly, CCL uses the keyword expose to specify this concept.




```
assembly Inner()(E){
        assume{
                E:Clock clock();
                E:Log log();
        }
        Comp c1(),c2();//instances
        //interactions
        clock:src1 ~> c1:snk1;
        c1:src2 ~> c2:snk2;
        c1:src3 ~> log:listen;
        c2:src5 ~> log:listen;
        expose {c2:sr4 as ext1}
}
```

Figure 4.20 Example of exposing assembly pins with CCL

Final important structural concept that needs to be covered is the assembly aggregation, i.e. assemblies within assemblies. This is done in a way very similar to components and is shown in figure 4.21.




```
assembly Outer1()(E){
        assume{
                E:Clock clock();
                E:Log log();
        }
        //instances
        Comp c3();
        Inner inner(){
            Inner:clock=clock;
            Inner:log=log;
        };
        //interactions
        inner:ext1 ~> c3:snk3;
        c3:src8 ~> log:snk5;
        expose {}
}
```

Figure 4.21 Example of specifying assemblies within assemblies

4.7.2. Behavioral elements of specifications

In addition to the structural specification of a component, which describes how a component can interact with the environment, CCL is also used to describe the internal behavior of a component. Component behavior description in CCL describes primarily how a component reacts to stimulus from the environment, which is the only possible source of stimulus. A component cannot stimulate itself, i.e. it cannot initiate behavior on its own.

Component behavior in CCL is defined through the use of a state-chart language based on UML state-charts. An executable action language has been added to the state-chart language for CCL purposes. However the graphical state-chart notation is not included in the development environment. Following are two differences between CCL and State charts that are important to mention:

CCL uses a subset of concepts defined in the UML State charts, most significantly there are no hierarchical states and no concurrent sub-states (within a reaction) in CCL.

CCL defines more specific semantics for UML elements that are identified as „semantic variation points“ (e.g., the queuing policy for events queued for consumption by a state machine). These more specific semantics are the result of the Pin component technology execution semantics.

Following are some fundamental concepts important in understanding component behavior in CCL.

Component behavior in CCL (reactions) is described in a language based on state-charts. Basic building elements of state-charts are states and transitions between states. Therefore states and transitions between states are formalized in CCL. Figure below shows an example of a simple state-chart.

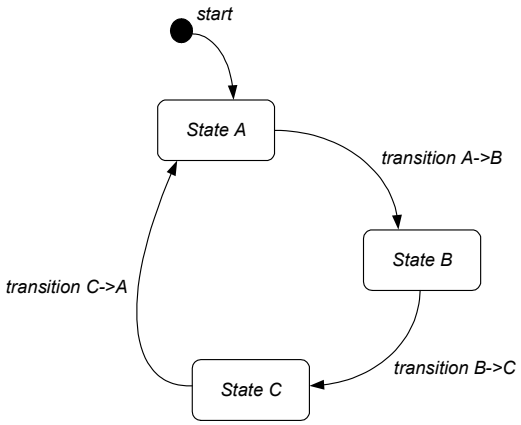


Figure 4.22 State-chart describing simple component behavior

Since component behavior consists of reacting to external stimuli, behavior of a component is described through the concept of a reaction. Reactions specify how a component will behave in response to stimulus from the environment on its sink pins, and what the components response, on its source pins, will be. Reactions are formalized in CCL through the keyword react.

Therefore, the components environment stimulates the component through its sink pins. This stimulation is considered an event which induces a reaction in the component. The ultimate purpose of the reaction is to produce some response on components source pins, again in the form of an event.

A reaction begins when an event has occurred at the sink pins, and its purpose is to produce an event on the source pins. Two kinds of events are possible at each pin: start events and end events. Start events are those that begin interaction with a component, and end events are those that terminate interactions. Start events are formalized in CCL by the ^ operator, followed by the respective pin name, for example: ^toc means a start event on toc pin. End events are formalized in CCL by the $ operator, also followed by the respective pin name, for example: $tic represents a end event on tic pin.

CCL imposes a well-formedness rule which states that each sink pin must be associated with exactly one reaction. The result of this is no ambiguity concerning the components response to external stimulus. The rule for source pins is such that each source pin must be associated with at least one reaction. This enables the possibility that multiple reactions might interact with the same external resource that's connected to a particular source pin.

Specification of a reaction contains:

- specification of states
- specification of transitions between states
- optional variable declarations

States and transitions make up the overall structure of a state-chart describing the reaction. States are formalized in CCL using the keyword state. Transitions are specified in terms of two states: stateA_ID -> stateB_ID{...}. The transition body between the curly brackets contains a more detailed transition specification. There is a special state called start which is the initial state a reaction is in.

The actual actions performed by the reaction (component) are specified using a action language that is a integral part of CCL. For each state it is possible to define a set of actions to perform on entering the state (using the keyword enter), and also a set of actions to perform when leaving a state (using the keyword exit). For each transition it is possible to define a action that is executed when a transition occurs, and this is defined inside the transition body.

It is not necessary to define in advance all the states in a reaction. States can be implicitly defined through transitions, which are of course obligatory to be specified.

Specification of a transition (transition body) contains:

- triggers (optional)
- guards (optional)
- actions (optional)

Triggers enable specification of events that initiate the execution of a transition. There are two types of triggers: event triggers and time triggers. Event triggers fire on pin events: ^pin (beginning of communication) or $pin (end of communication). Time triggers fire after a defined period of time has elapsed. Triggers are formalized through the keyword trigger. Time triggers use the additional after keyword which accepts an argument equal to the amount of time that needs to pass before the trigger can fire.

```
environment E() {
   service Clock() {
     source unicast src();
     threaded react ticking(src){
       start->sleeping{}
       sleeping->pulsing{
         trigger after(10);
         action ^src();
       }
       pulsing->sleeping{
         trigger $src();
       }
     }
   }
  service Log() {
    sink asynch snk();
    threaded react
       logging(snk) {
     start->run{}
     run->run{
       trigger ^snk();
       action $snk();
     }
   }
  }
}
```

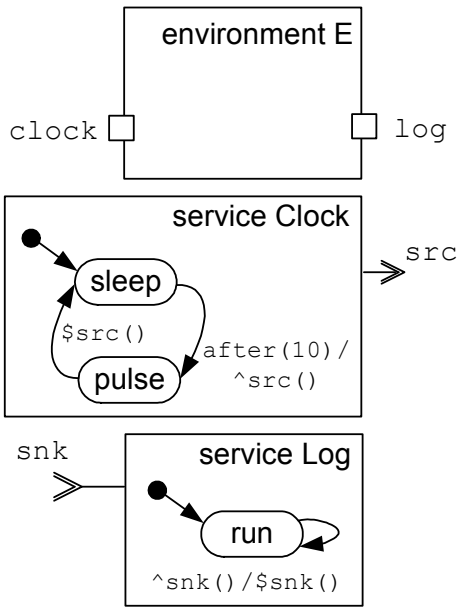


Figure 4.23 Example of using triggers in a reaction

Guards are specific conditions that need to be fulfilled before a transition can happen. Guards are essentially boolean expression on variables and pin events.

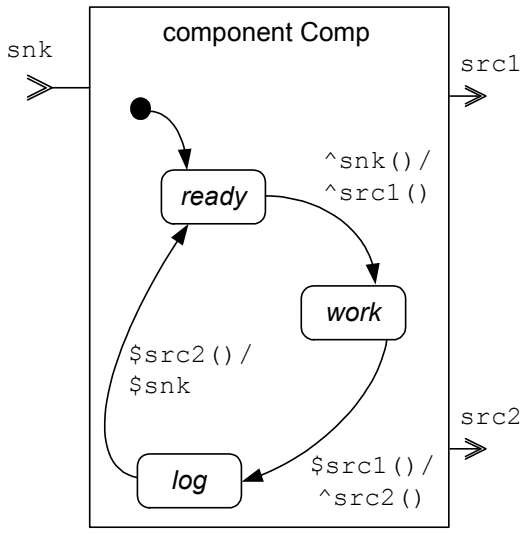
Actions defined for a transition execute when a transition fires. Actions enable almost standard programming language options, such as assignments, iterations, if-else blocks, performing pin events etc. This is equal to the actions that can be performed on entering or leaving a specific state.

Several well-formedness rules are defined for transitions. One rule is that transitions in a reaction may only be triggered on ^sink_pin events (where sink_pin is the name of a sink pin) or on $source_pin events (where source_pin is the name of a source pin). This means that only the initiation of communication on a sink pin may trigger a transition. This rule has its dual, which means that only $source_pin and ^sink_pin events may be executed as actions. Semantics behind this are reasonable. The rules say that a reaction in component A cannon force component B, which initiated communication with A, to stop communicating with A. Only component B can end this communication, therefore A can only observe the termination of communication.

An important issue regarding components reactions and their execution is the threading policy provided by CCL. Reactions can be declared threaded or unthreaded which is the default. The nature of a CCL thread is such that it represents primarily a unit of concurrency and not a concrete thread of execution, such as a thread in a programming language.

In the figure below is an example of a simple component with one reaction. The reaction is specified both in graphical and textual form. CCL provides formal support only for textual description of reactions, whereas the graphical representation through state-charts is not formally supported in the language nor in the PACC development environment.

Structural aspects of the component, as described earlier, are clearly visible both in the graphical and textual specification. The component in the figure has one reaction passIt which is graphically represented inside the component as a state-chart. The state-charts CCL counterpart is shown on the right.




```
component Comp(){
   sink asynch snk();
   source unicast src1();
   source unicast src2();
   threaded react passIt(
         snk, src1, src2) {
      start->ready{}
      ready->work{
         trigger ^snk();
         action ^src1();
      }
      work->log{
         trigger $src1();
         action ^src2();
      }
      log->ready{
         trigger $src2();
         action $snk();
      }
   }
}
```

Figure 4.24 Example of a complete CCL reaction

In the figure above, the passIt reaction is initially in the ready state. The transition to the work state requires a trigger in the form of an communication start event on pin snk. When this event occurs the transition will fire, and execute its defined action, i.e. it will begin communication through the source pin src1, and the reaction will achieve the work state. Next transition from the work state to log state will occur when the communication on the src1 port terminates, and the transition will begin communication on the src2 pin. Finally when communication on the src2 pin also terminates, the reaction will begin the transition back to the ready state, and terminating communication on the snk pin as well. Described behavior clearly demonstrates that this component passes data from its input to its output, hence the reaction name.

4.8. CCL specification transformation

CCL transformation procedure is generally made up of two processes. In the first part, model checking, the CCL specification is transformed into a C/FSP equivalent, where safety, security and other policies are verified using a model checker (Copper). In the second part, certified source code generation, the specification is transformed into a Pin/C equivalent and finally into a executable binary.

CCL transformation procedures are basically a part of the overall component certification procedure. Component certification in CCL is based on two paradigms:

- CMC (Certified Model Checking) and
- PCC (Proof Carrying Code).

The CMC procedure is based on checking the validity of finite state models against policies expressed in temporal logic. The finite state models are generated from the CCL specification into a C/FSP form. The temporal logic used is SE-LTL. FSP (Finite State Processes) is a simple algebraic notation used for describing process models. Every FSP description has an associated state machine. The CMC procedure is automated and is able to cover a broad range of policies (including safety and liveness). A drawback is that CMC is only able to verify source code such as C.

On the other hand, PCC works on the machine-code level and has the purpose of proving that specific machine code respects given policies. The proof is packaged together with the binary code so that it can be independently verified. PCC therefore operates only on binaries and has been limited to checking simple memory safety policies.

The problem with both procedures is that they generate relatively large proofs.

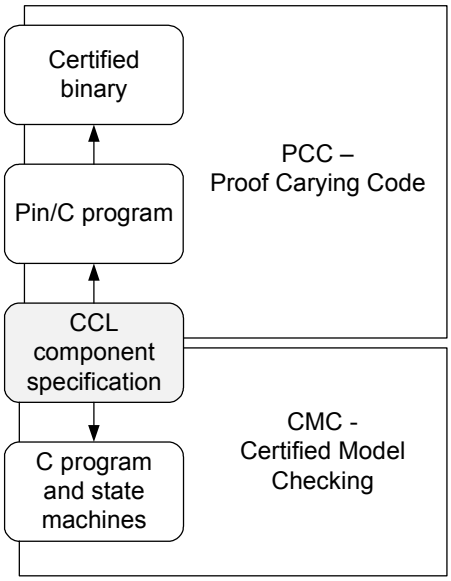


Figure 4.25 Overview of CCL transformation procedures

4.9. Certified model checking

The C/FSP form is generated first, directly from the CCL specifications, and it contains both the logical behavior specified by the state-chart language as well as infrastructure rules defined by Pin. There are a number of other restrictions on the generated C code (no internal concurrency, all variables are integral type, no pointers are allowed, etc.). Given that CCL already has a number of built-in restrictions these rules are not problematic.

Policies that need to be verified are expressed in CCL specification as SE-LTL formulas. The policies are of course also transformed, and their resulting form are equivalent Buchi automatons.

The generated C/FSP form serves as input to Copper, a state-of-the-art certifying software model checker that uses theorem provers (TP) and boolean satisfiability solvers (SAT). Copper will produce either a counterexample (CE) to the desired policy , or a ranking function, i.e. the proof that the specification conforms to the given policy.

In the C/FSP form of the CCL specification, every state of the state machine described in the specification is realized with a standard program block. CCL language guards are replaced by IF statements, state transitions are replaced by GOTO statements etc. This transformation and thus the equivalence between the CCL form and the C/FSP forms are straightforward and formally defined. Implementation of inter-component communication and of annotations used for reverse interpretation from C/FSP to CCL are less intuitive compared to transition-goto analogy but are also formally defined.

Inter-component communication in the model checker (Copper) is realized through FSP's event semantics. Message-based (asynchronous) interaction between Pin components is realized through FSP events. As described earlier, in Pin, inter-component communication is realized either in synchronous or asynchronous mode. Beginning and the ending of communication are defined in CCL syntax through symbols ^ (for communication start) and $

(for communication end). These concepts are mapped to FSP events in the transformation process from CCL to C/FSP.

A single begin event (^pin) is represented by the copper_handshake() function. A more complicated situation is when a component has a choice between multiple beginning events. This choice is implemented through FSP event synchronization. Since this is not described in C in a simple manner, FSP's specification facilities are used for this purpose. A call is made to the fsp_externalChoice() function, and a specification of the function behavior as a FSP process is given. This allows a choice among multiple events and a integer indicating the chosen event is returned.

The second nonstraightforward element of the transformation are the annotations used for reverse interpretation of the C/FSP program back to CCL. These annotations are realized through calls to ccl_node(x) function calls. Parameters passed to the function are nodes in the abstract syntax tree of the CCL specification that correspond to C statements following the annotation. The calls to this function are eliminated from the C/FSP program prior to verification, but Copper includes them in the final verification result (ranking functions).

The final step in CMC is to reverse-relating the generated ranking function back to the original CCL specification. This is done by mapping elements from the interpreted C program back to CCL elements. When CMC is finished, if a component satisfies all of its policies, the evidence of that is a ranking function expressed as nodes of the abstract syntax tree for the component's CCL specification. Therefore final proof that a component satisfies the specified policies is a ranking function (generated by Copper) expressed in the form of abstract syntax tree nodes of the components CCL specification.

4.10. Certified source code generation

Generation of certified source code begins with the CCL specification of the component and the generated ranking function. Software tools for Pin/C code generation from CCL specifications exists within the PACC project. This code generator was extended in order to support generation of certified code. The generator embeds invariants from the ranking function into the generated Pin/C code. An important decision in the transformation design process was how to embed this information into the generated code.

Method of embedding invariants into the code consists of making two function calls before the location used by each invariant. Invariant is then used as the argument to the second function call. After the compilation of this code, generated assembly code contains recognizable assembly call instructions, and instructions required to represent invariants are located between these calls. The Pin/C code generator was upgraded to insert these calls whenever invariants were provided by the ranking function.

```
                    __begin__();      1:     bl __begin__
                                      2:     li %r0,0
                                      3:     stw %r0,16(%r31)
                                      4:     lwz %r0,8(%r31)
                                      5:     cmpwi %cr7,%r0,0
                                      6:     blt %cr7,.L5
                                      7:     lwz %r0,8(%r31)
                                      8:     cmpwi %cr7,%r0,9
                                      9:     bgt %cr7,.L5
                                      10:    li %r0,1
                                      11:    stw %r0,16(%r31)
                                         .L5:
                                      12:    lwz %r3,16(%r31)
                                      13:    crxor 6,6,6
__inv__((n > =0) && (n < 10));        14:    bl __inv__
```

Figure 4.26 Example of the invariant embedding method

Finally, the resulting Pin/C source code includes all the invariants necessary to generate a proof that the binary component satisfies given policies.

```
else if (_THIS_->R_CURRENT_STATE == 1) {
   __begin__();
   __inv__(((__pcc_claim__ == 0 && __pcc_specstate__ == 8 &&
       pcc_rank__ == 0 && ((-2 < _THIS_->R_i ) &&
       (_THIS_->R_i != -1 ) && (_THIS_->R_i < 7 )) &&
       _THIS_->R_CURRENT_STATE == 1))); /* 52 */

if (pMessage->sinkPin == 0 /* ^incr */ ) {
...
```

Figure 4.27 Example of the final Pin/C code transformed from CCL

4.11. Certified binary generation

Certified binary code is the final product of the PACC process. It is generally made up of two parts: the binary itself and a certificate, i.e. the proof of the verification condition.

The certified binary is generated by simply compiling the source code using a standard compiler. The generated binary will contain assembly instructions, specifically calls to __begin__() and __inv__(...) functions. The assembly code between these calls is called a binary invariant.

Construction of the certificate begins by constructing the verification condition VC which is performed for one binary invariant after another. A verification condition is generated for each binary invariant, thus the overall verification condition will be a conjunction of all the individual verification conditions. From the overall verification condition, the certificate is obtained by proving the VC using a SAT-based theorem prover.

Once the binary and the certificate are produced, the binary is validated by checking that the certificate is a correct proof of the validity of the verification condition. Validation will be successful if and only if the certificate is a proper proof of the verification condition. When

the certified binary has been validated, the embedded binary invariants are removed from the final product.

4.12. CCL status and its future

Primary development force for CCL has been the exploration of what constitutes a general language like CCL, rather then trying to rigorously syntactically define CCL. Much information has thus been gathered about the former, at the expense of having less syntactical elegance in CCL than would be generally expected. A particular area that will need to be looked into in much more detail are component connectors which are not yet fully stabile in CCL.

As far as the language semantics are concerned, two approaches are considered. One approach suggests using structured operation semantics, while the other one suggests a solution like creating a UML profile for CCL (or extending the UML metamodel to embrace CCL). Both approaches have their advantages and drawbacks. Main advantage in favor of the structured semantics approach is the possibility to easily develop transformations towards different model checkers and development environments. The UML approach offers a better integration of CCL with UML environments.

Generally, the language is still under considerable development.

5. Applying CCL to ProCom

5.1. PACC and ProCom general comparison

ProCom is a software component model within the Progress project. The goal of Progress is to provide theories, methods and tools to increase quality and reduce costs in the development of vehicular, automation and telecommunication systems [10]. Therefore the target domain for ProCom are embedded software systems with high security, safety and efficiency requirements. Progress is intended to offer solutions for the entire development process, from a vague description to a final and precise specification ready for deployment. All of this, however, is still in early development phase. This is also true for ProCom which does not yet have a functional implementation in the form of a component model API, supporting runtime environment etc. In addition to modeling with components (which is the topic of this paper), PROGRESS puts a strong emphasis on analysis and deployment.

CCL is a construction and composition language used for creating structural and behavioral specifications of component-based systems. CCL is an essential part of the PECT (Prediction Enabled Component Technology) approach to component-based development that has been developed as part of the PACC initiative. The goal of the PACC initiative is to investigate technologies and methods for enabling reliable prediction of runtime behavior of component assemblies from their certifiable properties. The PECT concept has been developed as part of the initiative as a mean for achieving that goal. At the highest level, PECT is a scheme for systematic and repeatable integration of software component technology and design analysis and verification technology The PACC project states that achieving full reliable prediction is the only feasible use for the successful application of component-based development in software development. The general target domain for PACC are high-stakes embedded systems in diverse technology areas where performance, safety and security are of significant business value.

Regarding the target domain, there is high compatibility between Progress and PACC projects. The focus of PACC research has been on systems with highly deterministic, periodic, and reactive behavior, and property theories that are either verifiable (e.g., model checking) or present a clear falsification strategy (e.g., latency). PACC therefore focuses on control-intensive systems with low data manipulation, that are often found in various industrial, power and other systems. Progress is focused on embedded systems that typically have to function under severe resource limitations in terms of memory, bandwidth and energy, and often under difficult environmental conditions (e.g. heat, dust, constant vibrations). Progress recognizes the increased requirements for real-time behavior, meaning that a system must react correctly to events in a well-specified amount of time, i.e. neither too fast nor too slow.

Both projects are aimed at providing tools and methods for the entire system development process. PACC has gone much further in this respect then Progress. Tools and methods provided by PACC have been successfully used in a real prototype implementation of a substation automation system in association with ABB, where the PECT concept has proven its value in developing predictable assemblies for various types of power system controllers.

Progress emphasizes the importance of analysis during the system development phase and after the system has been deployed. Predictability is thus present in Progress in the context of predicting the values of certain system properties at runtime. Thus to achieve predictability, Progress relies on analysis to provide estimations and guarantees of different important properties. The analysis is present throughout the whole system development process and the analysis results are dependant on the level of completeness and accuracy of the component models and system design description. Early and inaccurate analysis may be performed during design to guide design decisions and provide early estimates. After development the analysis may be used to validate that the created component assemblies meet the original requirements. The different analyses planned for PROGRESS include reliability predictions, analysis of functional compliance (e.g. ensuring compatibility of interconnected interfaces), timing analysis (analysis of high-level timing as well as low-level worst-case execution time analysis) and resource usage analysis (e.g. memory, communication bandwidth).

The PACC project takes the position that achieving reliable predictions of assembly behavior is the only feasible use for the successful application of component-based development in software development. PACC relies on mathematical methods and theories to provide basis for predictions of different system properties. In current software development practice predictability is achieved on a system-by-system basis and primarily through rigorous testing. This notion of predictability relies on the premise that if enough past executions are observed, future behavior can be predicted. Taking into account all the familiar limitations of software testing, a far better approach, that is taken by PACC, is to develop analytic theories that predict the behavior of entire classes of systems. PACC assumes the usage of both logical (safety, liveness, etc.) and empirical theories (latency, etc.). The focus in the PACC project is on developing analytical theories for system properties of significant business value such as: reliability, safety, security etc. Future work will extend the focus to systems exhibiting increasingly stochastic behavior (e.g., behavior sensitive to the distribution profiles of stimuli) and property theories whose falsification strategies are not inherently clear (e.g., reliability theories based on statistical testing of component reliability).

The PACC project takes a much more formal and firm approach to predictability then Progress. Unlike Progress, PACC takes the position that full certifiable predictability is the only feasible way for successful utilization of CBD. Progress on the other hand makes no such assumption but does acknowledge the importance of predicting system properties. This difference can probably be explained by the significant difference in the development level between the two projects, and it is likely that Progress might, when further developed, adopt a position similar to the position of the PACC project.

5.2. Application strategies

5.2.1. ProCom PECT development

Given the nature of the PACC project and the PECT concept, primary application strategy of ProCom would be to develop a ProCom PECT. The PACC project identifies and describes a PECT development process for arbitrary component technologies. General process consists of four phases:

Definition – functional and statistical requirements of the PECT are defined, assembly property to analyze is determined

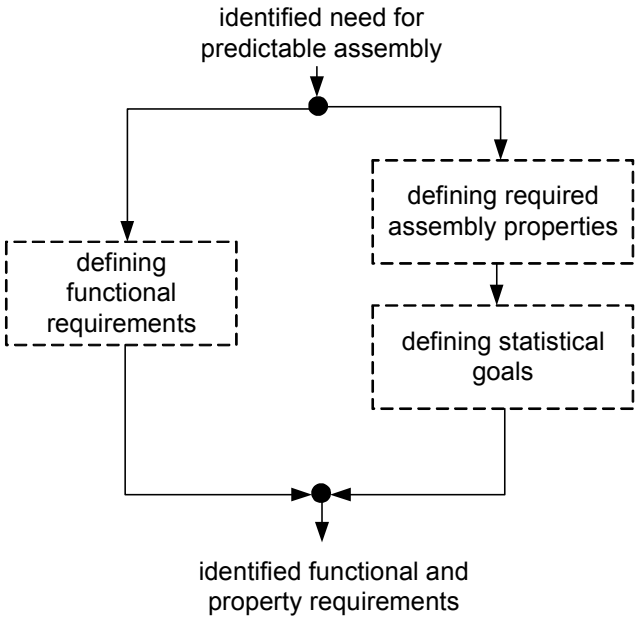


Figure 5.1 PECT definition phase workflow

Two parallel paths exist: one to define the functional requirements and another to gather the necessary requirements for the property of interest and set the PECT's prediction goals. Properties can be any kind of a value that we are interested in, such as latency etc. Developing support for a property means developing the analysis support for it, i.e. the reasoning framework. Considering that reasoning frameworks then impose design constraints, this step must define what kind of changes must be done in the component model in order to supply components with required information about the observed property. Functional requirements define the range of assemblies that must be possible to produce by the construction model and the tolerance requirements define the desired quality of predictions based on the analysis methods supported by the PECT.

Considering that the component model exists (at least on paper), this phase would not be as complex while developing a ProCom PECT. Additional time should be invested in analyzing the properties of assemblies that need to be observed and predicted. ProCom defines support for describing component attributes that might provide a basis for component information required by reasoning frameworks, however this must be developed much further.

Co-Refinement – creation of component and analysis models with a PECT instance as a result of this phase

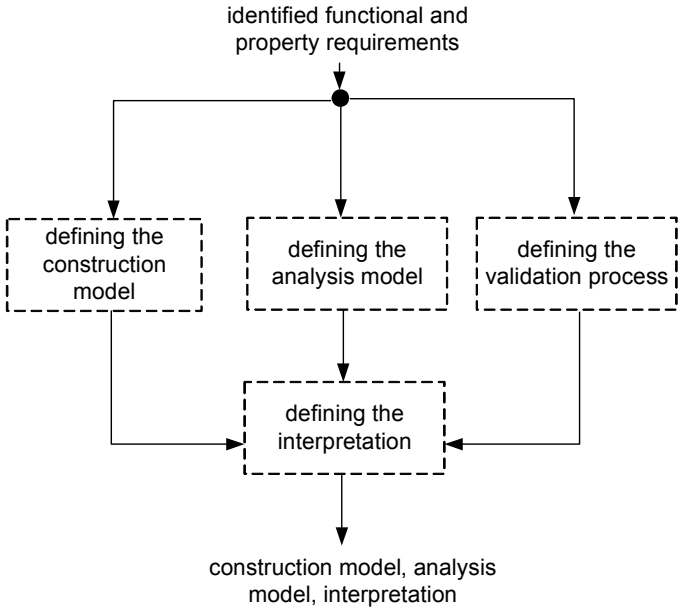


Figure 5.2 PECT co-refinement phase

Co-refinement is basically a process of iterative negotiation between the constructive and analysis points of view. The constructive point of view pushes for assembly generality; the more assemblies that can be represented in the construction model, the better. The analysis point of view pushes for predictability, which implies constraining the construction model to adhere to the assumptions of the property theories that enable analysis and prediction. These forces tend to act in opposition to each other. [6]. From a practical point of view it is important to achieve only a level of generality that is necessary to represent some class of realistic systems, therefore clear bounds must be set on generality. It is also important to set limits to the complexity of the analysis model, because it is expected that the analysis can be done in reasonable time with reasonable computing resources. It is very important that the construction model is transformable to the analysis model, which is enabled by the interpretation.

The co-refinement process starts with an initial description of the "languages" for the constructive and analysis models. The elements of the construction model language (CCL is an example of such a language) are influenced by the PECT's target domains and how systems in those domains will be structured, developed, deployed, and sustained (evolved) over time. The elements of the analysis model language are simply a subset of some property theory that is suitable for analyzing the behavior of interest. [6]. During the development of the ProCom PECT, a dialect of CCL (ProCom CCL) might be developed that is customized to the specific requirements of the ProCom component model.

A real PECT development workflow might not be as structured and formalized as the one shown in the figure above. Although all of the processes are shown to be purely concurrent, they are actually significantly intertwined and dependant on each other. During the co-refinement phase, the reason for iteration (different then the latter validation iteration) is an assessment of whether the resulting constructive assembly is general enough to handle the required range of assemblies and whether the property theory supporting the analysis model

will, with reasonable computation and interpretation effort, give predictions for all these assemblies.

The component model is, to a large extent, defined in ProCom. It would be required to develop a property theory for the desired property of assemblies and if needed to adapt the model according to the theory. In addition to the component model, as explained earlier, the construction framework within a PECT contains a abstract component technology, that uses a construction language for specifying assemblies of components within the PECT. For a PECT based on ProCom, CCL could be used as a constructive language, but with certain adjustments. Behavioral part of the language could most probably remain the same, but the structural part of the language would have to be adjusted to the ProCom structural definitions, given that CCL is currently adapted to the Pin component model that differs from ProCom from a structural point of view. A more detailed analysis of the language adjustment is done in the next section where the strategy of adjusting CCL to ProCom is analyzed.

Validation – PECT is validated against the defined goals

Validation generally aims at showing that the designed PECT gives the expected results in term of predictions it makes on given assemblies. When considering empirical property theories (i.e. theories based on measurement) validation goal is stated in terms of the probability that a prediction will lie within some accuracy bounds, with some stated confidence level. Validating the accuracy of a prediction is analogous to validating a scientific theory. That is, behaviors are observed systematically under controlled circumstances, and this observed behavior is then compared to predicted behavior. The key word is systematic, since validation results should be, above all else, objective and hence repeatable. While collecting validation data it is important to have good laboratory techniques so that anomalies can be studied and results can be repeated. Validation data is finally analyzed which has a twofold purpose: first, to objectively and reliably describe the predictive powers of a PECT; second, to provide analysis data to support additional co-refinement, should normative goals fail to be satisfied [6]

Packaging – if the PECT instance satisfies the goals it is packaged and delivered

One objective is to ready the technology for deployment, including the development of installation support, documentation, and so forth. A second and more fundamental objective is to design automation support for the PECT to minimize the property-theory-specific expertise required by PECT users to make effective use of analysis models supported by the PECT. [6]

5.2.2. Adaptation of CCL to ProCom

Information that CCL specifications provide can be divided into three general groups:

- structural information
- behavioral information
- analysis specific information.

The structural part of the component specification specifies how the component interacts with its environment. CCL structural elements correspond to the Pin component model, since CCL and the prototype PECT it belongs to were created for developing systems ultimately realized in Pin. Therefore in CCL the only way a component interacts with its environment is through its pins. There are no other ways of communication with the component. The structural specification defines for example the type of communication with the environment, type of data exchanged with the environment etc.

This corresponds to ProCom where the components external, i.e. structural, interface consists only of ports which handle all communication with the component. All the functionality provided by the component is thus accessed through ports, which corresponds to the Pin model where pins handle the same kind of responsibilities.

In Pin/CCL there are two types of pins: sink pins and source pins. The component receives stimulus for communication only through its sink pins and it initiates communication with the environment only through its source pins. Therefore primary classification of pins is done according to the direction of communication with the component, i.e. towards the component through sink pins, or from the component through source pins.
ProCom uses the port concept for handling communication with the component which is analogous to a pin in CCL. Ports are also primarily classified by direction, thus there are input and output ports. Input ports of a component form the input port group and the output ports form the output port group.
The sink pins of a Pin/CCL component correspond to the input port group on a ProCom component. The input port group is accordingly responsible for handling the communication coming towards the component. Analogously source pins of a Pin/CCL correspond to the output port group.

The external structural differences between ProCom and CCL components can be resolved by adapting the CCL grammar to the specific details of the ProCom model. New language constructs, accustomed to ProCom, must replace the existing ones. It is primarily important to enable differentiation between data and trigger ports. For this purpose two new keywords must be introduced: data and trigger. A practical solution would be to require the usage of only the trigger keyword for trigger ports, whereas a port would be considered a data port by default. In this way the usage of the data keyword would be optional. For defining the direction of communication, two new keywords: in and out would be introduced. They are self-explanatory. Considering the semantics of ProCom ports, trigger ports would not receive any parameters, which would be required for data ports.
Considering that ProCom doesn't specify details on communication mechanisms between components, no adequate alternatives can be determined for the type of communication normally supported by CCL (synchronous or asynchronous).

New ProCom CCL constructs	**Replaced CCL construct**
trigger in	sink
trigger out	source
data in (in)	sink
data out (out)	source

Table 5.1 Structural element changes in ProCom CCL

There is high compatibility between primary elements of functionality in components, i.e. reactions in CCL and services in ProCom. ProCom components are intended to implement small and low-level functionality therefore services within a ProCom component will not perform extensive and complex operations. There can be more than one service within a ProCom component therefore functionality of a ProCom component is completely made available to external users through a set of services. Each service can be triggered independently of the others and multiple services may run concurrently. Each service has a single input port group which contains exactly one trigger port and a set of data ports. Each service also has at least one output port group where the data produced by the service is made available. Allowing multiple output groups provides the possibility to produce outputs at different points in time. A service may also have attributes attached to it.

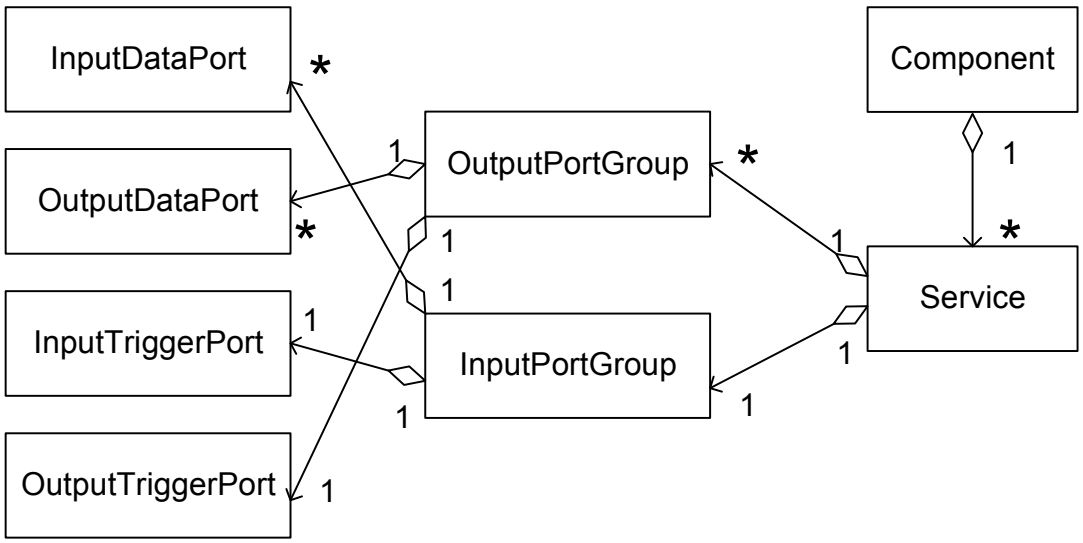


Figure 5.3 Relation of ports to services in ProCom components

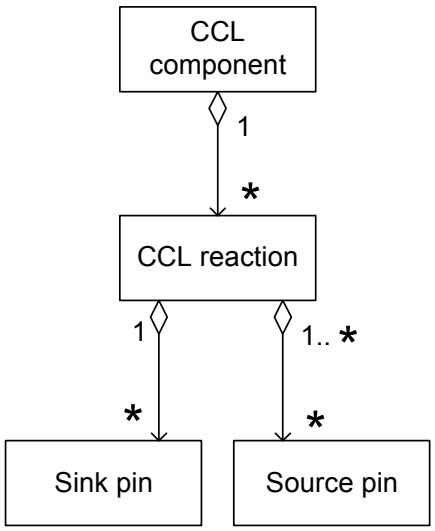


Figure 5.4 Relation of pins to reactions in CCL components

A CCL reaction is highly compatible with a ProCom service. It is a unit of functionality within a CCL component. The relation between CCL pins and components reaction is similar to the relation between ProCom ports and services. There are certain differences regarding ways how a reaction or a service can be stimulated. Both ProCom services and CCL reactions are initially in a passive state. ProCom services are activated by a signal on their trigger port. They read the data on the data ports, perform the computation, set the resulting data on the output data ports and signal the receiving component through the output trigger port. Considering this behavior, it is clear that only stimulus that a ProCom service can receive from the outside is the initial stimulus which initiates execution of a service. All the computation that the service performs from then on can in no way be influenced by the component external environment. Therefore looking from the outside, a ProCom service only changes between two states: active and passive. Internally the service may go through multiple states depending on the nature of its calculations but this is not visible from the outside.

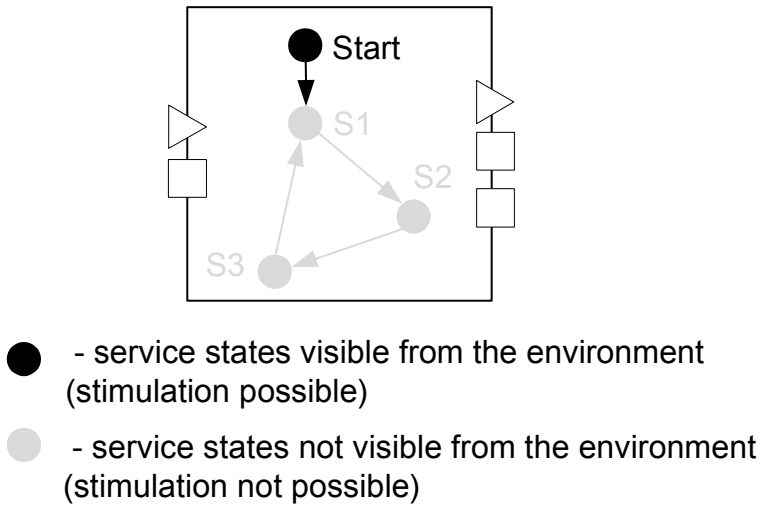


Figure 5.5 External accessibility of ProCom service states

Behavior of a ProCom service described above is somewhat different from the behavior of a CCL reaction. A CCL reaction is also started through the beginning of communication on one of the corresponding sink pins. The reaction may then, before it finishes its calculations, also go through a series of states. Main difference from ProCom is that the transitions between these states can be triggered by the component external environment. Therefore the components environment in CCL has a much larger influence on component behavior then in ProCom. This was described in the CCL section, where the details of the action language were discussed. Considering that CCL offers a broader set of possibilities than is required by ProCom, the behavioral part of CCL can be easily exploited for modeling the behavior of a ProCom component. The development environment and the compiler will ensure that only a subset of CCL is utilized which semantically corresponds to the nature of ProCom components and services.

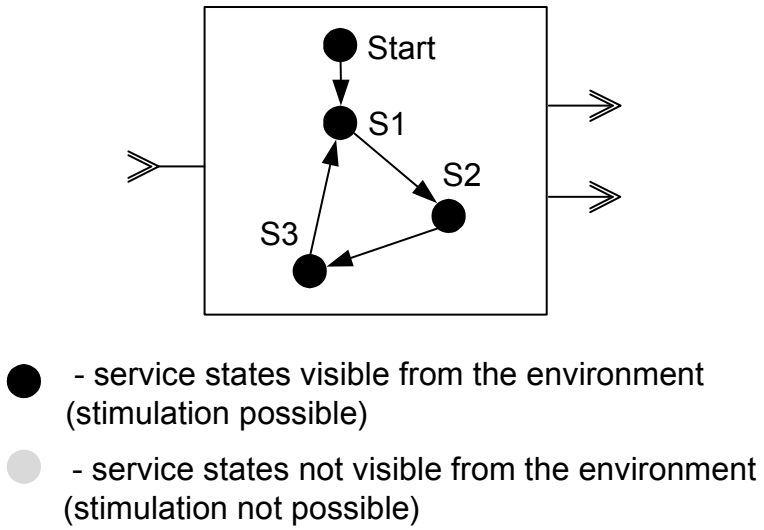
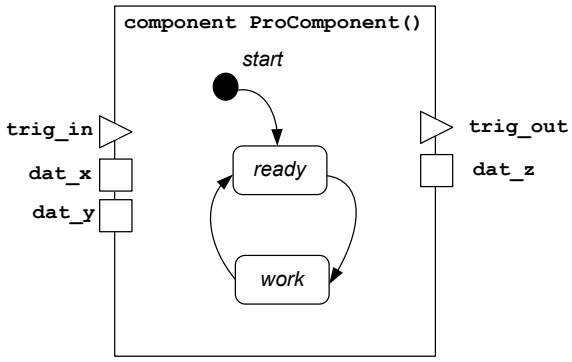


Figure 5.6 External accessibility of CCL reaction states

In CCL a reaction began when an event occurred at the sink pins. Two kinds of events are possible at each pin: start events and end events. Start events are those that begin interaction with a component, and end events are those that terminate interactions. Start events are formalized in CCL by the ^ operator, followed by the respective pin name, for example: ^toc means a start event on toc pin. End events are formalized in CCL by the $ operator, also followed by the respective pin name, for example: $tic represents an end event on tic pin. Considering that ProCom components do not support the level of communication supported by CCL, the $ operator will be discarded and only the ^ operator will be used to describe the activation of the trigger port (both input and output).

Considering the limited accessibility of ProCom service states compared to CCL, it is obvious that the triggering abilities for firing transitions among states in a reaction, must be limited only to trigger ports and only for the initial transition from the passive to the active state. It must be possible only to begin the execution of a service through an input trigger port, and to activate the output trigger port once the service has stopped executing. This also means that triggering must be disabled for all states except the initial state. Since external triggering is therefore unable to be used for transitions between states, other CCL constructs may be used. The only remaining construct is a guard over some condition which can be tested to determine if a transition is eligible to fire. The CCL trigger keyword will therefore allowed to be used only for the transition from the passive state to the initial service state. This transition was done implicitly in CCL but in ProCom it might be done explicitly. Regarding the action part of the language, which defined what the component “can do” in a certain state or when a transition fires, all the internal component calculation functionality should remain the same except for the port communication options. Only communication action allowed in the ProCom-CCL would be the activation of the output trigger port. This activation will only be enabled in the final service state, after all the computations were completed and the data ports set. After the output trigger has been activated the service returns to the passive state.




```
component ProComponent(){

      trigger in trig_in();
      data in dat_x(int x);
      data in dat_y(int y);
      trigger out trig_out();
      data out dat_z(int z);

      threaded react simpleWork (
            trig_in,dat_x,dat_y,trig_out,dat_z) {

            boolean gate = false;

            state work{
                  dat_z = dat_x + dat_y;
                  gate = true;
            }

            start->ready{}

            ready->work{
                  trigger trig_in;
            }

            work->ready{
                  guard gate==true;
                  action{
                        trigger trig_out;
                        gate = false;
                  }
            }
      }
}
```

Figure 5.7 Example of a simple ProCom CCL behavior model

6. xUML – Executable UML

6.1. Overview of UML and MDA

Modeling is a proven and a well-accepted engineering technique. Before building a software system for example there is a need of an overall architectural model of the system, as well as models describing other important aspects of the system, which will serve not only to the developers who will actually implement the system, but also for the client who are paying for the system etc. Essential about a model is that it shows what a system is supposed to do and not how it is supposed to be done. Therefore models usually do not contain many technical implementation details, they are concerned with the overall design. So models actually represent a simplification of reality. Models are also built because complex systems can't usually be understood entirely from a single viewpoint. Therefore developing an appropriate model, or a set of models, for a system is highly important because it can influence how a problem is attacked and how the solution is shaped.

Unified Modeling Language (UML) is a standardized specification language for object modeling of systems. It is a general-purpose modeling language that offers both a textual syntax and a (now nearly ubiquitous) graphical notation used to develop models of systems. UML was created by the OMG 1997. A UML model of a system is typically made up of several diagrams that are built using UML's graphical notation symbols. The language defines rules for combining symbols into diagrams. So what is essentially gained by creating different diagrams are multiple views on a system, each addressing a specific concern. Each model is a semantically closed abstraction of a system. A model is usually a combination of these diagrams. So UML is used in a very broad sense to visualize, specify and document software systems.

To understand the conceptual model of the language it is necessary to understand three major language elements: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML. There are three kinds of building blocks: things (first-class abstractions in a model), relationships (way to tie things together) and diagrams (groupings of interesting collections of things).
Things are further divided into: structural things (class, interface, collaboration, use-case, component, etc.), behavioral things (interaction, state-machine, etc.), grouping and annotational things.
There are four kinds of relationships in UML: dependency, association, generalization and realization. A dependency is a semantic relationship between two model elements in which a change to one element (the independent one) may affect the semantics of the other element (the dependent one). An association is a structural relationship among classes that describes a set of links, a link being a connection among objects that are instances of the classes. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. A generalization is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent). A realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. [11]
Most important language elements are of course diagrams. A diagram is the graphical presentation of a set of elements, usually in the form of a connected graph of vertices (things)

and paths (relationships). Diagrams are created to visualize (model) a system from different perspectives, so a diagram is a projection into a system. In theory, a diagram may contain any combination of things and relationships. In practice, however, only a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software system. Therefore, UML offers thirteen kinds of diagrams: class diagram, object diagram, component diagram, composite structure diagram, use case diagram, sequence diagram, communication diagram, state diagram, activity diagram, deployment diagram, package diagram, timing diagram, interaction overview diagram.

Only a brief description of some of the diagrams is given here, but considering that the focus of the thesis is on modeling the behavior, most important diagrams for this will be covered in much more detail in the next sections. UML component diagrams are not of interest here since they are only concerned with modeling the static structure view of a system.

A class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. A component diagram is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors. Component diagrams address the static design implementation view of a system. A state diagram shows a state machine, consisting of states, transitions, events, and activities. A state diagrams shows the dynamic view of an object. An activity diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation. Activity diagrams address the dynamic view of a system.

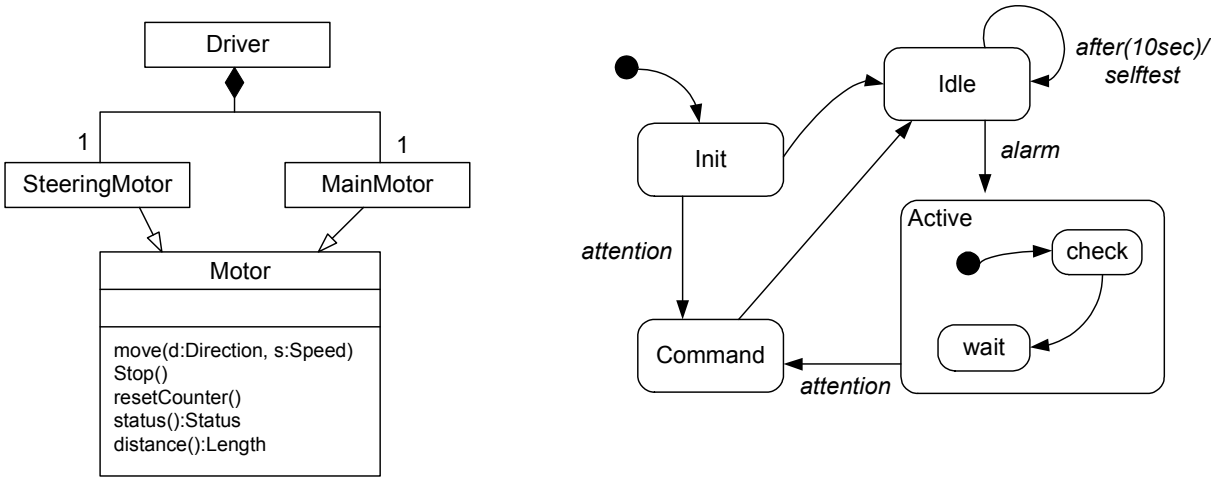


Figure 6.1 Class diagram and state machine diagram examples

UML and xUML are the main supports of the Model-Driven Architecture (MDA) initiative announced by the Object Management Group (OMG) in early 2001, the purpose of which is to enable specification of systems using models. Model-driven architecture depends on the notion of a Platform-Independent Model (PIM), a model of a solution to a problem that does not rely on any implementation technologies. A PIM is independent of its platform(s). A PIM

can be built using an executable UML. Because an executable model is required as a way to specify PIMs completely that xUML is a solid foundation of model-driven architectures.
MDA also defines the concept of a Platform-Specific Model (PSM): a model that contains within it the details of the implementation, enough that code can be generated from it. A PSM is produced by weaving together the application model and the platforms on which it relies. The PSM contains information about software structure, enough information, possibly, to be able to generate code. Executable UML views the PSM as an intermediate graphical form of the code that is dispensable in the case of complete code generation. [11]

MDA, as defined by the OMG, essentially calls for the mapping of a Platform Independent Model (PIM) onto a Platform Specific Model (PSM). A PIM is a complete application specification that is independent of the technology platform upon which it will eventually execute. PIMs map onto PSMs, which provide the systems architecture infrastructure, or "plumbing," that implements the PIM on a specific technology platform, turning it into an executable application. One way to view this is to think of the PIM as source code for a solution, the mapping function as a compiler, and the PSM as an execution environment. The term Virtual Execution Environment (VEE) conveys a sense of the ultimate intent of the PSM. [13]

OMG has started to provide standard PIM-to-PSM mappings for many of the popular technology platforms such as CORBA, J2EE, and .NET. Along with these mappings other OMG standards such as the Meta-Object Facility (MOF), XML Metadata Interchange (XMI), and the Common Warehouse Metamodel (CWM) work in concert to make the MDA a complete and robust approach to software development. The MOF provides a standard repository for UML models and defines a structure that allows a common meta-view of the stored UML models. XMI allows companies to exchange UML models as streams or files with a standard format based on XML. The CWM standardizes how to represent database models, schema transformation models, OLAP, and data mining models. [13]

To become executable, the modeled application, or PIM, depends upon the architectural services of the PSM. The UML profile used to create the executable model places specific requirements upon the PSM. In other words, each UML element that is part of that profile must be somehow supported via the mapping into an element of the PSM that can be part of the execution environment. The mapping itself is typically referred to as model compiling and the mapping mechanism a model compiler. The main purpose of the model compiler is to translate each modeled element of the PSM into an element that can be executed within the architectural framework of the VEE.

MDA is still however under considerable development.

6.2. Overview of xUML - executable UML

The primary purpose for the development of xUML was the need for achieving a higher level of semantic precision when developing system models, which was not available in UML. So it was the mission of xUML to define the semantics of different subjects in sufficient detail so the created model of system behavior could be executed. Therefore xUML aims at a lower abstraction level then UML itself, but still a higher level when compared to object-oriented programming languages. It is not a programming language itself, i.e. it does

not aim at making any decisions on the code level. From the MDA point of view, xUML enables creating of Platform Independent Models (PIM's).

UML version 1.x was not executable because it was semantically incomplete and ambiguous. With the introduction of UML 2.0 in 2001. with action semantics, UML became executable. Action semantics provided a complete set of actions at a higher level of abstraction. This was added to UML through a Action Specific Language which defined in a precise way the action semantics of the language. Also many semantically weak elements from the version 1.0 of the language were remove. All of this resulted in the newly enabled executability of UML. xUML is a UML profile and is thus completely based on fundamental UML elements and its extensibility mechanisms.

The xUML process is considered to be a rigorous object-oriented system development method which is based upon the principle of building a set of precise testable analysis models of the system to be developed, executing defined tests on these models and defining a systematic strategy by which the models will be used to produce code for the desired target system. [7]

The xUML process embodies the following characteristics:

- Precise, complete analysis models that can be tested using simulation.
- Simple notations that can be understood by a wide audience
- Clear separation of subject matters (system domains)
- Useable analysis models without ambiguous interpretations
- Implementation by translation in which the entire system can be automatically generated from the analysis models
- Large-scale reuse

An important benefit from introducing UML was the possibility to reduce subjectivity when designing a system, and introduce formal descriptions of what is expected of the system. Any decisions that are made during the development process can be aligned with the behavior that the system clearly exhibits, and whether this behavior meets the requirements. Another very important improvement was the possibility to detect many errors very early in the design process. This is important because the cost of correcting an error increases as the project advances.

The history of software development is a history of raising the level of abstraction. As technology moved from one language to another, generally the level of abstraction at which the developer operates was increased, requiring the developer to learn a new higher-level language that may then be mapped into lower-level ones, from C++ to C to assembly code to machine code and the hardware. Knowledge of an application was then formalized in as high a level language as it could be. Over time, developers learned how to use this language and applied a set of conventions for its use. These conventions became formalized and a higher-level language was born that was mapped automatically into the lower-level language. In turn, this next-higher-level language was perceived as low level, and a set of conventions for its use were developed. These newer conventions were then formalized and mapped into the next level down, and so on.

xUML is at the next higher layer of abstraction comparing to the current state of software development techniques, abstracting away both specific programming languages and decisions about the organization of the software. Therefore a specification built in xUML can be deployed in various software environments without change. Physically, an xUML

specification comprises a set of models represented as diagrams that describe and define the conceptualization and behavior of the real or hypothetical world under study. The set of models, taken together, comprise a single specification that can be examined from several points of view. There are three fundamental projections on the specification, though any number of UML diagrams can be built to examine the specification in particular ways. [11]

Concept	Known as	Modeled as	Formalised using
all the things in the world	data	classes attributes associations	UML class diagram
lifecycles of things	control	states events transitions	UML statechart diagram
what things do	algorithm	actions	action language

Table 6.1 Concepts in an Executable UML Model [11]

A typical first step would be to create a primary class diagram to clearly describe the subject matter at hand (system domain). Each object must be subjected to and conform to the well-defined and explicitly stated rules or policies of the subject matter being analyzed, attributes must be abstractions of characteristics of things in the subject matter being analyzed, and that relationships similarly model associations in the subject matter.
The objects (instances of the classes identified previously) may have lifecycles (behaviors over time) that are modeled as state machines. The state machines are defined for classes, and expressed using a UML state chart diagram. The behavior of the system is driven by objects moving from one stage in their lifecycles to another in response to events. When an object changes state, something must happen to make this new state be so. Each state machine has a set of procedures, one of which is executed when the object changes state, thus establishing the new state.
Each procedure comprises a set of actions. Actions carry out the fundamental computation in the system, and each action is a primitive unit of computation, such as a data access, a selection, or a loop. The UML only recently defined semantics for actions, and it currently has no standard notation or syntax, though several near-conforming languages are available.
These three models—the class model, the state machines for the classes, and the states' procedures—form a complete definition of the subject matter under study. [11]

It is important to explain in more detail what “executable” in xUML really means. UML versions prior to version 2.0 were not executable and they provided for an extremely limited set of actions (sending a signal, creating an object, destroying an object, etc.). In 2001, the UML was extended by semantics for actions. The action semantics provides a complete set of actions at a high level of abstraction. For example, it became possible to write actions for manipulating collections of objects directly (avoided the need for explicit programming of loops and iterators). Another very important thing required for UML to become executable, are rules that define the dynamic semantics of the specification. Dynamically, each object is thought of as executing concurrently, asynchronously with respect to all others. Each object may be executing a procedure or waiting for something to happen to cause it to execute.

Sequence is defined for each object separately; there is no global time and any required synchronization between objects must be modeled explicitly. The existence of a defined dynamic semantics makes the three models computationally complete. A specification can therefore be executed, verified, and translated into implementation using a model compiler. [11]

6.3. Transforming xUML – model compilers

The most important aspect xUML is the possibility to execute UML models. As explained earlier, xUML raised the level of abstraction when compared to today's programming languages such as Java. Unlike Java, it offers a new development technique, which abstracts much of the technical details, by creating a set of models of a system being developed. All these models together form a semantically consistent and functional whole which can, using a adequate compiler, be transformed to the next lower level and thus made executable. Therefore it is safe to argue that xUML is another (graphical) programming language. Since a xUML model completely specifies the semantics of a certain system domain, it really in a way becomes a program of that system domain. But much less details are required to be defined when compared to a lower language. In fact, xUML models deal exclusively with the "objects" of the system domain, and completely abstract the technical details such as: classes and objects, distribution of programs into threads, storage of data etc. All this issues are considered related to specific software and hardware platforms. The way these technical decisions are abstracted away in xUML can be compared to the way register and heap issues are abstracted from today's programming languages such as Java. So in the same way modern language compiler would make decisions about register allocation on its own, so would a xUML compiler make decisions concerning a particular hardware and software environment and making decisions such as whether or not to use a distributed Internet model, to use separate threads for each user window etc.

An executable UML model compiler turns an executable UML model into an implementation using a set of decisions about the target hardware and software environment. There are many possible executable UML model compilers for different system architectures. Each architecture makes its own decisions about the organization of hardware and software, including even the programming language. Each model compiler can compile any executable UML model into an implementation. A single model compiler may employ several languages or approaches to problems such as persistence and multi-tasking. Then, however, the several approaches must be shown to fit together into a single, coherent whole.

Examples of possible model compilers: [11]

- Multi-tasking C++ optimized for embedded systems, targeting Windows, Solaris, and various real-time operating systems.
- Multi-processing C++ with transaction safety and rollback.
- Fault-tolerant, multi-processing C++ with persistence supporting three processor types and two operating systems.
- C straight on to an embedded system, with no operating system.
- C++, widely distributed discrete-event simulation, Windows, and UNIX.
- Java byte code for single-tasking Java with EJB session beans and XML interfaces.
- Handel-C and C++ for system-level hardware/software development.
- A directly executing executable UML virtual machine.

A system developer would then build a xUML model that captures his solution for the system domain under study, acquire a model compiler that meets the performance properties and system characteristics he requires, and give directives to the compiler for the particular application. In practice it is not yet this simple. Specific performance requirements would sometimes have to be looked out for. One particularly performance-sensitive feature is static allocation to tasks and processors. For example allocating two classes that communicate heavily with different processors could cause significant degradation of network and system performances. In such a scenario it would simply be required to re-allocate the elements of the model and recompile. This is where xUML demonstrates its power because by separating the model of the subject matter from its software structure, the two aspects can be changed independently, making it easier to modify one without adversely affecting the other. This extends the Java notion of the "write once, run anywhere" concept because as the level of abstraction is raised, the programs become more portable.

6.4. Overview of system development in xUML

When building an xUML model of a system, specific models are built for each domain (subject matter) in the system. It is therefore important to understand what domains actually are as well as to identify them in a system that's being developed. Similar to usual system development, requirements are gathered for the system and use cases are built from them. Requirement gathering and domain identification are two concurrent processes. Requirements are placed in corresponding domains according to their vocabularies. Typically the requirement gathering and domain identification are not completely separated but rather intertwined.

A domain is a autonomous world inhabited by conceptual entities. Important to emphasize is that conceptual entities in one domain require the existence of other conceptual entities in the same domain but not in other domains. Each domain, except for the overall application, provides services to other domains. Relationships between domains are given through a domain chart. The domain chart provides the highest-level view of the system.

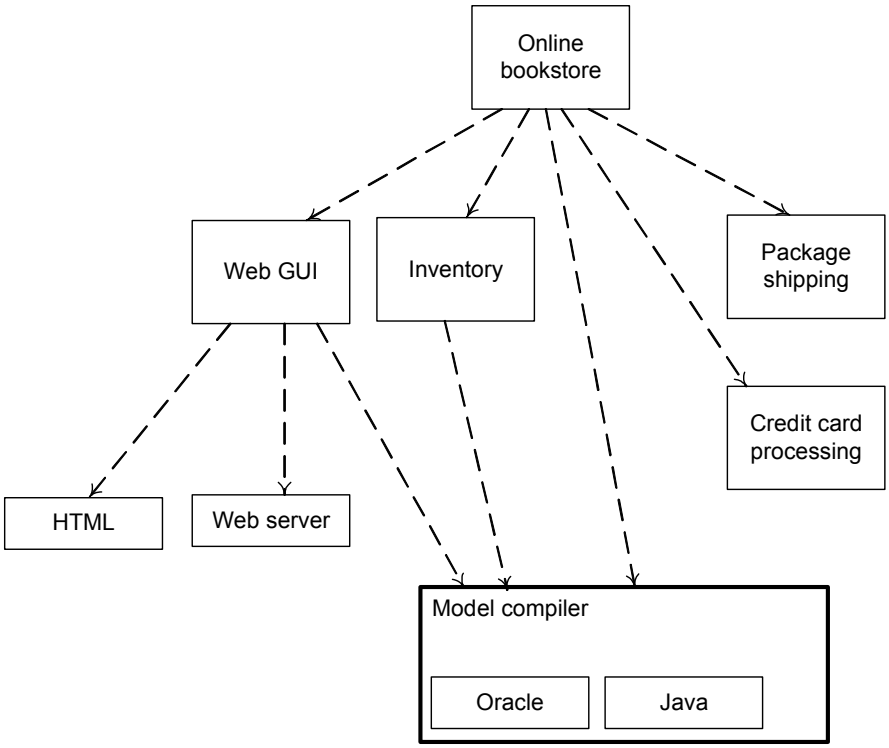


Figure 6.2 xUML domain chart example

Each package symbol on the domain chart represents a domain, and the dotted lines are domain dependencies. The example domain chart shows an online bookstore with a web GUI. The bookstore has no knowledge of the concepts within the web GUI, so it could be replaced by any other user interface. The functional or behavioral requirements on the system can be expressed textually or more formally in terms of use cases. Systems respond to requests from actors, and each collection of responses, an interaction, is a use case. Each use case is a set of behavioral requirements placed on the system by the role played by the actor. The vocabulary used in the expression of the use case should match the vocabulary of the associated domain. Hence, the use case "Select a book to purchase from a pull-down menu" would be acceptable if "book" and "pull-down menu" were both concepts in the same domain. [11]

After the domains have been identified and the overall requirements defined, it is possible to make detailed decision about the behavior of individual domains. To do this successfully, the developer must have a complete understanding of the domain being modeled. The xUML model for each domain will contain several different complementary diagrams such as class diagram, state diagrams etc.

Since a domain is usually full of different, important and unimportant, concepts, it is crucial to extract the ones that are important for the system being modeled, i.e. that correspond to the purpose of the domain. Result of this primary abstraction process is a class diagram containing classes, attributes, associations etc.

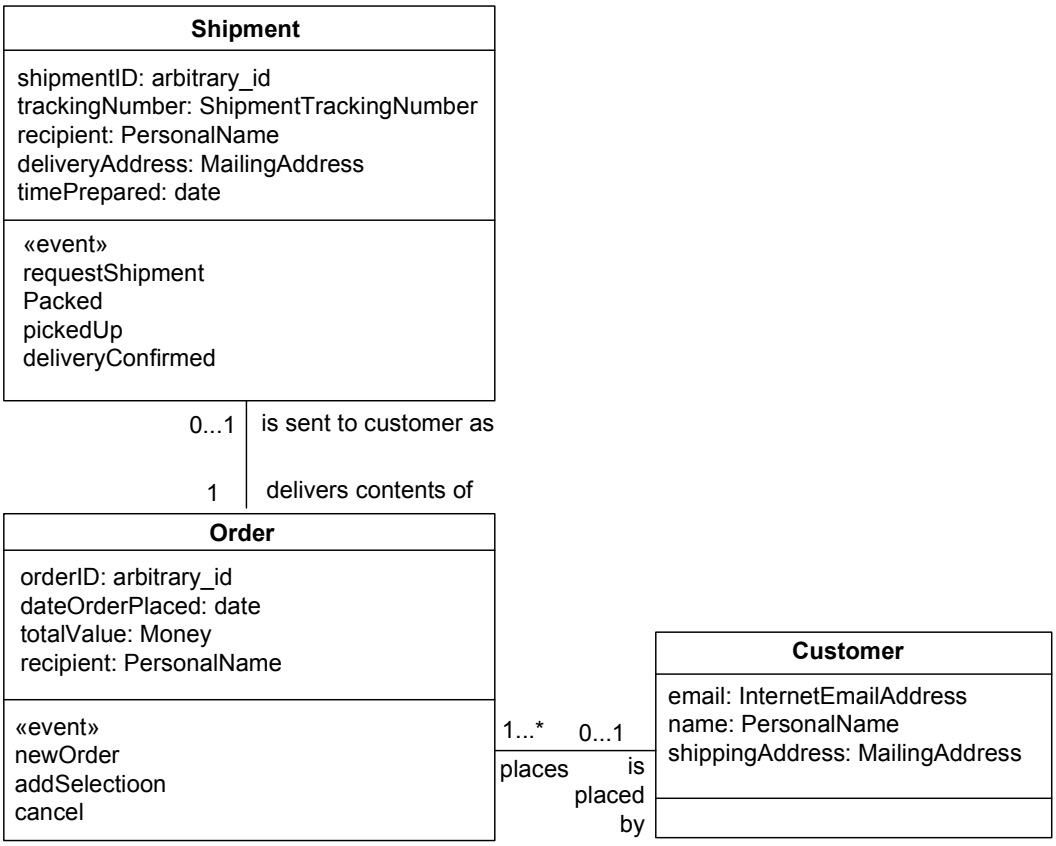


Figure 6.3 Example xUML class diagram

When the structure of the domain has been described using the class diagram, it is important to also create a very good description of domain behavior, i.e. behavior of its elements. All the things in the domain go through various lifecycles, i.e. a collection of stages or states. A state machine is a formalization of a life cycle. Concepts such as states, events, transitions and procedures are used to describe a life cycle of a thing through a state chart. Like things have a common lifecycle, so when a group of like things is abstracted as a class, the common lifecycle is abstracted as the object's state machine.

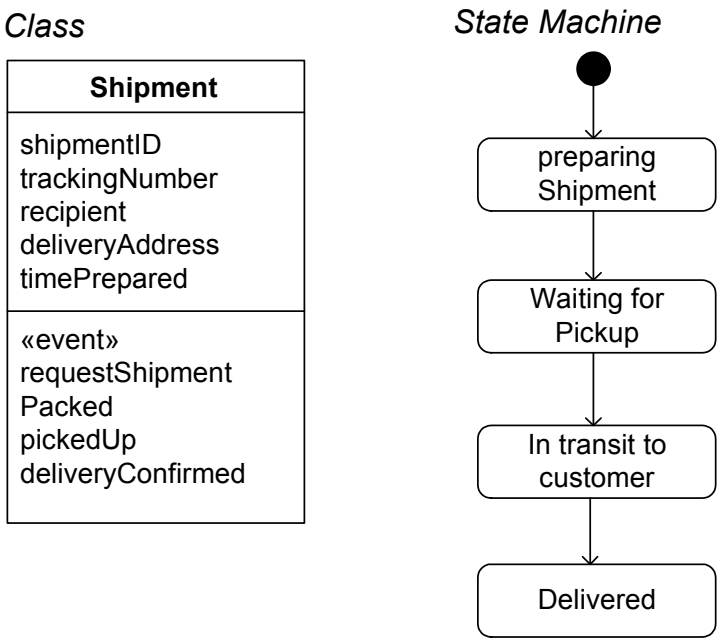


Figure 6.4 Example of a xUML state machine for a class of things [11]

The state machine is represented using a subset of the state chart diagram. This subset is chosen to be rich enough to model the lifecycles of the abstractions, in contrast to the more complex state chart diagrams required for modeling software structure. The subset is also chosen to be sparse enough to ease model compilation: A complex language requires more complex model compilers. [11]

Each state on the state chart diagram has an associated procedure that takes as input the data items associated with the event that triggered entry into the state. Each procedure comprises a set of actions, and each action carries out some functional computation, data access, signal generation and the like. Actions are like code, except at a higher level of abstraction, making no assumptions about software structure or implementation. UML has a definition of the semantics of actions, but it does not yet have a notation for action models. [11]

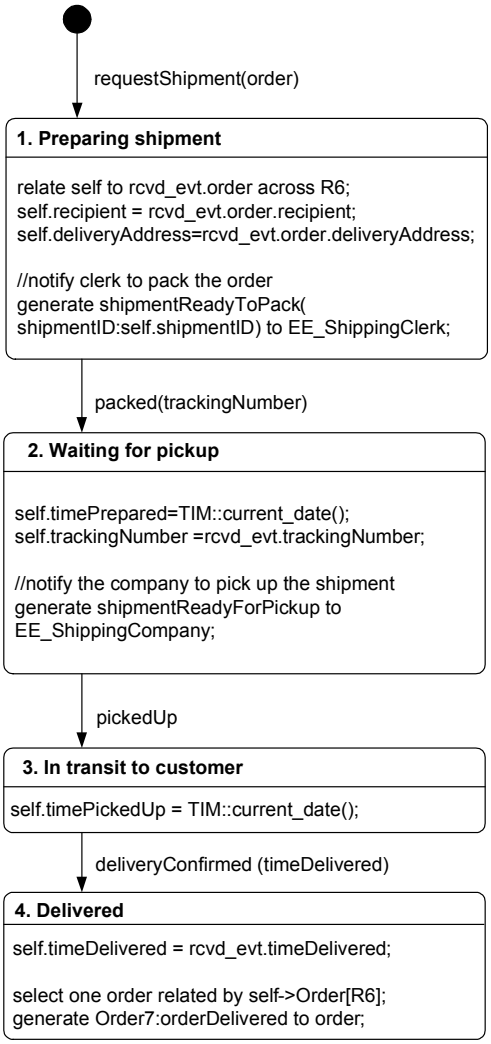


Figure 6.5 Example of a xUML state chart diagram with executable actions [11]

After adequate models have been built, the overall system model may be verified and compiled. Typically all the models and diagrams will not be perfectly done right away. It will be necessary to reiterate the modeling process and to reevaluate how the concepts were gathered in the class diagram, how the behavior was captured in the state chart diagram etc. An important rule is that classes should be kept as simple as possible. Altogether the entire development process will likely take multiple iterations before the final version, but this all depends on the complexity of the system being developed, experience of the developer, quality of available development tools, etc.

7. Applying xUML to ProCom

7.1. General applicability of xUML in ProCom behavior modeling

It is clear from the examples in the previous sections that xUML modeling serves primarily for creating highly abstract system models, without any technical detail regarding hardware and software platform. In that respect, xUML offers a higher abstraction then for example Java which is a purely object-oriented language. The xUML language can therefore, in its complete form, be used for developing highly complex systems, i.e. systems that can be primarily decomposed into a domain chart and where domains can further be detailed through the class diagram, state chart diagram, collaboration diagram etc. Any possible reuse would have to be realized at the level of the model compiler in the form of reusing technical implementations, since at this point it is difficult to assume how the reuse concept, which is fundamental to CBD, could be applied systematically and generically at the system and domain level.

Regardless of the reuse principle, primary objective here is to establish the applicability of xUML for modeling the behavior of an individual component. It has been clearly stated in ProCom that components provide small and compact service, i.e. functional units. It has also been said that component implementations will most likely be supplied in the form of source code (probably C). Therefore it is safe to assume that the functionality provided by the component will in itself be mostly of algorithmic nature and will not require object modeling. Another important argument that supports this conclusion is the simple difference in resource consumption of object-oriented programs in languages such as Java when compared to programs in lower level procedural languages such as C. It then follows that the functionality of a single component will not have a structure decomposable into domains and furthermore describable through class diagrams. Taking into account the algorithmic nature of a typical component service, only element that could probably be utilized for component behavior modeling are the state-chart diagrams and corresponding action languages. There are action languages that can be considered for behavior modeling, however their application in the component context, falls out of the xUML development process and must therefore be considered separately from that process.

As mentioned in the previous section, UML has a definition of the semantics of actions, but it does not yet have a notation for action models. However several action languages exist that are compliant with UML action semantics, such as:

- BridgePoint® Object Action Language
- SMALL - Shlaer-Mellor Action Language
- TALL - That Action Language

Because of the very simple nature of the functionality realized by a typical component, a small subset of these languages would typically be required to realize it. Most of the object-oriented functionality is discarded because it is not applicable. Therefore in the next section only the essential elements of the Object Action Language are analyzed and an attempt is made to provide a relation to the ProCom component model.

7.2. Application of Object Action Language in ProCom behavior modeling

The Object Action Language (OAL) is an action language that realizes the action semantics specification for UML as of version 1.5. The OAL is used to define the semantics for the processing that occurs in an action in a state chart diagram of a xUML model.

An OAL action can perform five possible processes which are [12]:

- data access
- event generation
- test
- transformation
- bridge and function

These processes are supported through [12]:

- control logic,
- access to data described by the class diagram,
- access to data from events which initiate actions,
- ability to generate event
- access to timers and current date and time

In a UML model there is no concept of a "main" function or routine where execution starts. The models are executed as interacting finite state machines executing concurrently. Any state machine, upon receipt of an event (from another state machine or from outside the system) may respond by changing state. On entry to the new state, a block of processing (an "action") is performed. This processing can in principle execute at the same time as processing associated with another state machine. Concrete details regarding the relation of concurrency and execution of different state machines depend on the nature of the software and hardware architectures used to implement the system.

The execution rules in OAL are as follows:

1. Execution commences at the first statement in the action and proceeds sequentially through the succeeding lines as directed by any control logic structures
2. Execution of the action terminates when the last statement is completed

As explained in the previous section, the functionality of a single component will not have a structure decomposable into domains and furthermore describable through class diagrams. Therefore all the OAL constructs related to class manipulation and relationship management between classes and instances can be discarded when considering an OAL subset for component behavior modeling. Discarded language constructs will not be mentioned in details but only a list of the overall discarded functionality is provided.

Discarded functionality related to class manipulation and relationship management: instance creation, instance selection, writing attributes and reading attributes, instance deletion, creating relationship instances, deleting relationship instances, creating event instances, instance selection through relationship navigation, operations, bridges and functions.

Language constructs that remain in the language are: control logic, event generation, event data access, unary operators, date and time operators, timers and arithmetical, logical and string operators.

Language control logic enables the use of standard constructs such as: if, else, for, while, break and continue. They will not be analyzed in further detail.

The OAL expression of an action has access to and can produce certain data items. The following data items are available to be read at the start of and throughout an action: constants, values of attributes of classes (this is discarded and thus not available but mentioned for the sake of completeness), supplemental data items carried by the event that initiated the action and local variables (created by statements within the action). These data items can be produced during an action: local variables, values of attributes of classes, supplemental data items to be carried by an event generated during the action

integer	string	unique ID	timer handle
real	date	instance handle	event instance
boolean	timestamp	inst. handle set	state

Table 7.1 OAL data types [12]

Considering all the constructs regarding class and relationship manipulation were discarded, variables will only be able to have the fundamental types from the figure above, but not these types: instance handle, instance handle set, unique ID and event instance.

The scope of a variable is defined as the block of code in which the variable may be accessed. A block of code can be the entire OAL for the given action, or it may be a <statements> block within a control logic structure. Each control logic structure contains at least one new scope. All variables that were accessible in the scope containing the structure are also accessible in the block or blocks contained by the structure, essentially causing the contained scopes to inherit variables from the parent scope. Any variables declared within a given control logic block fall out of scope when execution exits the block. Control logic structures may contain multiple scopes, either by repeated nesting of new structures or by using the elif or else constructs in an if structure. When nesting of control logic is used, each new structure defines a new scope. In an if statement, each elif or else structure contained within the if block defines a new scope, and each new scope inherits the scope of the block containing the if statement. [12]

An important language construct that remained in the language is the possibility to generate events. In the ProCom context, an event will happen when a component service has finished its calculations, has set the output data ports to the calculated values and has activated the output trigger port and thus notifying the next component about the data. In contrast to this, the initial process when a component is triggered through its input port is not considered an event here. It is assumed that the component will implicitly receive the values at the input data ports through its local variables and that this variable initialization will be performed automatically by the executing framework.

Final important element to mention is the overall position and usage of a OAL state chart in the component modeling context. When developing xUML systems models, it is necessarily done in a visual development environment where graphical notation of UML is used extensively because writing textual description of models would be counter-productive and would be in contrary to the visual appeal of UML. Therefore, when considering creating a OAL state chart that describes behavior of a component it would be necessary to use a visual modeling tool for several reasons. When modeling class behavior, it is considered that the domain changes its state when a method is called on an object instance. Signature (name and arguments) of this method determine the next state of the state chart describing the behavior of this object (class). Thus, a model compiler will implicitly establish the connections between states in the state-chart and there is no need for the developer to formalize the transitions between states. Therefore, a visual modeling tool would be necessary to use, or syntax to formalize the transitions must be developed.

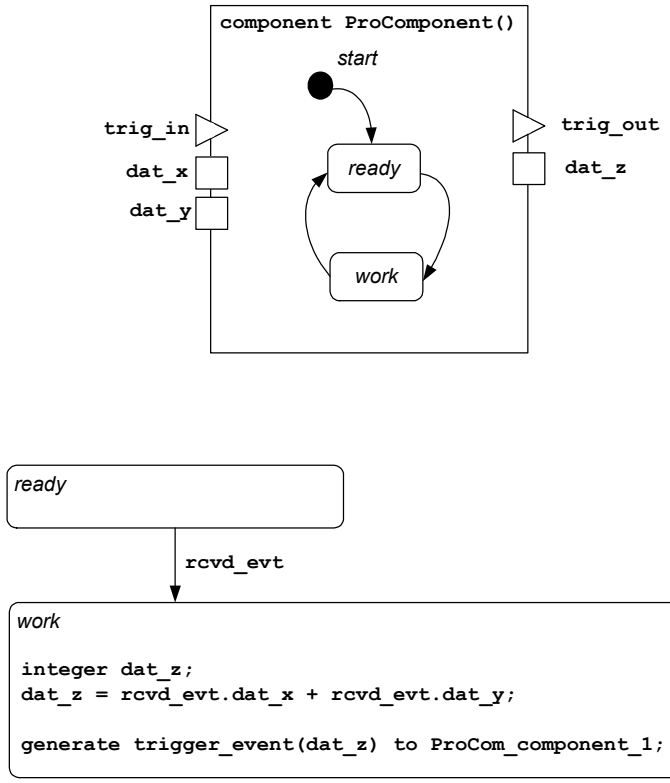


Figure 7.1 Example of a theoretical state chart for a ProCom component

8. Language summary

8.1. EP

EP is a declarative language for describing the behavior of platform-independent models (PIMs). EP is based on a hybrid notation that uses graphical elements as well as textual elements in the form of OCL code snippets. Compared to existing action languages it is more abstract. Much of the operation dynamics can be expressed using EPs graphical notation. EP differs from other OCL-based languages that enable the specification of only pre- and post-conditions, because it enables the development of an executable description of the dynamic behavior of the system which enables complete code generation. EP describes the dynamic behavior of a system by using events and properties from the class diagram. [16]

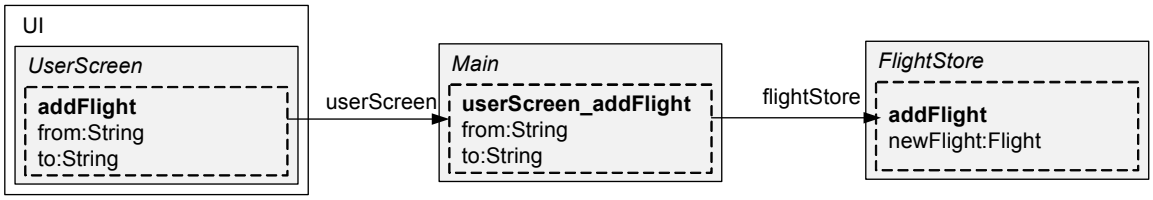


Figure 8.1 Example of a EP behavior model

8.2. ASL

ASL is an implementation independent language for specifying processing within the context of an Executable UML (xUML) model. ASL is compatible with the UMLs "Precise Action Semantics" extension. The purpose of the language is to provide an unambiguous, concise and readable definition of the processing to be carried out by an object-oriented system. ASL is a language providing: sequential logic, access to the data described by the class diagram, access to the data supplied by signals initiating actions, the ability to generate signals, access to timers, access to synchronous operations provided by classes and objects, access to operations provided by other domains, tests and transformations, etc. [17]

Unlike conventional languages, there is no concept of a main function or procedure where execution starts. ASL is executed in the context of a number of interacting state machines, all of which are considered to be executing concurrently. Any state machine, after receiving a signal (from another state machine or from outside the system) may respond by changing state. On entry to the new state, a block of processing is performed. This processing can, in principle execute at the same time as processing associated with another state machine. [17]

8.3. JUMBALA

Jumbala is an action language developed for the needs of an industrial project called SMUML (Symbolic Methods for UML Behavioral Diagrams) at the Laboratory for Theoretical Computer Science in Helsinki University of Technology, whose goal is to formally analyze behavioral aspects of reactive computer systems modeled in UML. The aim of the project is to build a prototype tool set for analyzing the behavior of industrial UML models using state-of-the-art symbolic model checking techniques.

The design of Jumbala as an UML action language conforms to the general requirements placed by the UML framework. Jumbala also conforms to the requirements made by SMUML. The key design principle has been to make Jumbala resemble the Java programming language. Most of the elements in Jumbala are taken directly from Java. Jumbala is roughly a subset of Java.[19]

8.4. Scrall

Scrall is an action language at the semantic level of relational action languages such as SMALL, TALL, OAL and ASL. All of these languages are platform independent. Scrall was primarily inspired by SMALL and exceed it in size. Scrall is primarily a graphical language. Symbols are networked together with object, data and control flows. Scrall is fully compatible with UML 2.0 action semantics. Scrall symbols are only about 10% consistent with standard UML graphics standards

8.5. TASM - Timed Abstract State Machine specification language

TASM is a behavior modeling language based on state charts that claims to successfully integrate functional and extra-functional properties on a system. The extra-functional requirements that TASM deals with are timing behavior and resource consumption. TASM is essentially an action language because its specifications are executable and the language has a defined execution and composition semantics.

It is based on the theory of abstract state machines. The abstract state machine formalism revolves around the concepts of an abstract machine and an abstract state. System behavior is specified as the computing steps of the abstract machine. A computing step is defined as a set of parallel updates made to global state. A state is defined as the values of all variables at a specific instant. A machine executes a step by yielding a set of state updates. A run, potentially infinite, is a sequence of steps. A basic abstract state machine specification is made up of two parts - an abstract state machine and an environment. The machine executes based on values in the environment and modifies values in the environment. [18]

9. Conclusion

CCL and PACC have proven to be very successful methods for developing predictable component-based software systems. The adapted action language (CCL) implemented in the Starter Kit has proven very well designed for its purpose, i.e. describing component-based systems and enabling their prediction and certification. An important fact is that all the PECT concepts have been successfully implemented and demonstrated in the case study with ABB. The PECT approach is thus highly promising for CBD.

Taking this into consideration, CCL has proven to be most convenient for modeling ProCom component behavior, since reasonable compatibility was determined between ProCom and the Pin component model utilized in the PECT that has been implemented in the Starter Kit. Therefore a language highly similar to CCL may be used in the future ProCom IDE, and its structure can be obtained from the CCL structure through a simple adaptation procedure. An attempt of such an adaptation is given in Appendix A but its validity remains to be proven, considering that none of the other structural elements were compared, such as assemblies and environments.

The alternative strategy, of developing a ProCom PECT has not been extensively analyzed but no obvious and significant problems have been identified that would disable the development of such an implementation.

Application of a typical xUML language was not concluded as feasible for the purpose of modeling component behavior. Reasons for this are high incompatibilities between the target domain areas, because xUML typically aims at modeling various object oriented systems that do not fall into the target domain of ProCom. Given the nature of xUML, and the fact that it is unreasonable to expect that a single language would give the solution to all software problems, a typical xUML development process is not feasible for this purpose. However one possible solution could be the utilization of the action (state chart) languages that are used in xUML for describing behavior of classes. Considerable adaptations must be done on such languages to increase their usability in the CBD area. Such adaptations to action languages would then inherently lead to a language very similar to CCL. CCL would therefore serve as a better starting point for the development of a ProCom modeling language then xUML action languages.

Only a few modeling languages were shortly summarized, considering that most of them fall into the xUML family and thus all the above conclusions apply to them as well.

10. Bibliography

[1] Crnković I., Larsson M.: Building Reliable Component-Based Software Systems, Artech House, 2002.

[2] Bachmann F., et al.: Volume II: Technical concepts of Component-Based Software Engineering, 2nd edition, CMU/SEI, 2000.

[3] Wallnau K.: Volume III: A Technology for Predictable Assembly from Certifiable Components, CMU/SEI, 2003.

[4] Wallnau K., Ivers J.: Snapshot of CCL: A Language for Predictable Assembly, CMU/SEI, 2003.

[5] Hissam S., et al: Pin Component Technology (V1.0) and Its C Interface, CMU/SEI, 2005.

[6] Hissam S., et al: Predictable Assembly of Substation Automation Systems: An Experiment Report, 2nd Edition, CMU/SEI, 2003.

[7] Chaki S., Ivers J., Lee P., Wallnau K., Zeilberger N.: Model-Driven Construction of Certified Binaries, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2007.

[8] Chaki S., Ivers J., Wallnau K.: Certified Binaries For Software Components, CMU/SEI, 2007.

[9] Wallnau K., Ivers J., Sinha N.: A Basis for Composition Language CL, CMU/SEI, 2002.

[10] Bureš T., Carlson J., Crnković I., Sentilles S., Vulgarakis A.: ProCom – the Progress Component Model Reference Manual, Mälardalen University, Västerås, Sweden, 2008.

[11] Mellor S., Balcer M.: Executable UML: A Foundation for Model-Driven Architecture, Addison-Wesley, 2002.

[12] Object Action LanguageTM Manual v1.4, Project Technology, http://www.ooatool.com/docs/OAL02.pdf

[13] Marta M., Witz G.; Executable UML and iUML, Computers and components, 2007.

[14] Sommerville I.: Software engineering 8th ed.: Addison-Wesley, 2007.

[16] Kelsen P., et al.: A Declarative Executable Language based on OCL for Specifying the Behavior of Platform-Independent Models, http://wiki.lassy.uni.lu/se2c-bib_download.php?id=2468

[17] Wilkie I., et al.: UML ASL reference guide, Kennedy Carter, 2003. http://www.ooatool.com/docs/ASL03.pdf

[18] Ouimet M.: Timed Abstract State Machines: An Executable Specification Language for Reactive Real-Time Systems, Embedded Systems Laboratory, Massachusetts Institute of Technology, http://esl.mit.edu/tasm/ESL-TIK-00193.pdf

[19] Dubrovin J.: JUMBALA — An action language for UML state machines, Helsinki University of Technology, 2006.

[20] Booch G., Rumbaugh J., Jacobson I.: The Unified Modeling Language User Guide 2nd ed., Addison Wesley, 2005.

Abstract

Software development today is faced with responding to huge challenges because of the increasing requirements on software systems in all domains, from the smallest embedded system, desktop and business software to large scale industrial systems. An approach to software system development that is based on the reusability principle and offers solutions to many crucial problems is component-based software engineering (CBSE) where systems are built by composing independent, tested and trusted software components.

Considering that the component is the fundamental building block of software systems in component-based development (CBD), it is reasonable to investigate and consider methods for modeling components functionality, i.e. the services it provides since modeling has today become a mainstream engineering technique. ProCom is a component model intended to be used for development of embedded systems that are both resource limited and control intensive.

This thesis analyzes Construction and composition language (CCL), a successful modeling language adapted to CBSE, and Executable UML (xUML) which represents a large family of modeling languages based on UML. Possibilities of applying these languages to modeling the behavior of ProCom component are analyzed and a model is provided for such applications.

Sažetak

Razvoj programskih sustava se danas nalazi pred velikim izazovima jer mora odgovoriti na rastuće zahtjeve koji se postavljaju na takve sustave u svim područjima primjene, od najmanjih ugradbenih sustava, kućnih i poslovnih aplikacija do velikih industrijskih sustava. Programsko inženjerstvo temeljeno na komponentama je pristup razvoju programskih sustava temeljen na konceptu ponovnog korištenja implementirane funkcionalnosti, gdje se sustavi grade kao kompozicije nezavisnih, isprobanih i vjerodostojnih programskih komponenti.

Obzirom na važnost komponente kao osnovnog gradivnog elementa takvih sustava, važno je istražiti metode modeliranja funkcionalnosti koje komponenta ostvaruje, tj. usluge koje pruža okolini. ProCom je komponenti model čija je svrha primarno omogućiti razvoj ugradbenih sustava s ograničenim resursima i intenzivnim upravljačkim svojstvima.

U ovom radu analizirani su Konstrukcijski i kompozicijski jezik (CCL), uspješni jezik za modeliranje, prilagođen CBSE procesu, te Izvršivi UML (xUML) koji predstavlja široku obitelj akcijskih jezika za modeliranje temeljenih na UML-u. Analizirane su mogućnosti primjene tih jezika za modeliranje ponašanja ProCom komponenti te je predložen model za ostvarenje takve primjene.

Biography

Ivan Ferdelja was born on January 6th 1984. in Zagreb, Croatia. In 2002. he graduated from Technical school "Ruđer Bošković" in Zagreb. In the same year he started his studies at the Faculty of electrical engineering and computing in Zagreb where he graduated in 2009. His personal interests include software development in the Java programming language as well as development of software for small and embedded computer systems.

Appendix A: Adapted CCL grammar for ProCom behavior modeling

Note: only component part of the grammar is given (syntax for assemblies and environments is excluded).

```
Start : cclSpec

cclSpec:
      topLevelUnit*

topLevelUnit:
      simpleUnit | constructiveUnit

constructiveUnit:
      assemblyDecl | componentDecl | environmentDecl

componentDecl:
      component ID ( [formalParam (, formalParam)*] ) { comPart* }

formalParam: [ const ] typeSpecifier [ & ] ID

comPart:
      annotation
      | declaration
      | triggerPortSpec
      | dataPortSpec
      | reaction
      | verbatim

triggerPortSpec:
      trigger (in | out) ID ( [portParam (,portParam)* ] ) ;

dataPortSpec:
      data (in | out) ID ([portParam (,portParam)* ]) ;

reaction:
      [threaded] react ID (pinXRef ) { reactStm* }

pinXref:
      { ID (, ID)* }

reactStm:
      declaration
      | stateSpec
      | transitionSpec

stateSpec:
       state ID  { action* }

action:
      verbatimLit
      | assignmentExpression ;
      | returnStatement ;
      | iteration
      | ifThenElse
      | alert ( conditionalExpression , conditionalExpression ) ;
      | { action action* }
```

```
iteration:
        whileIteration
        | forIteration
        | doWhileIteration

whileIteration:
        while ( conditionalExpression ) action

forIteration:
        for (
         [ assignmentExpression ] ;
         [ assignmentExpression ] ;
         [ assignmentExpression ] ) action

doWhileIteration:
        do action while ( conditionalExpression ) ;

returnStatement:
        return [ assignmentExpression ]

ifThenElse:
        if ( conditionalExpression ) action [ else action ]

transitionSpec:
        transitionHead -> ID { [event] [guard] [action] }

transitionHead:
        start | ID

event:
        | trigger eventTrigger ;
        | trigger timeTrigger ;

eventTrigger:
        unaryExpression Note: ID of a trigger port

timeTrigger:
after ( additiveExpression )

guard:
        guard conditionalExpression ;

assignmentExpression:
        conditionalExpression
        | unaryExpression = assignmentExpression

conditionalExpression:
        equalityExpression
        | conditionalExpression || equalityExpression
        | conditionalExpression && equalityExpression

constantExpression: conditionalExpression

equalityExpression:
        relationalExpression
        | equalityExpression == relationalExpression
        | equalityExpression != relationalExpression

relationalExpression:
```

```
        additiveExpression
        | relationalExpression < additiveExpression
        | relationalExpression <= additiveExpression
        | relationalExpression > additiveExpression
        | relationalExpression >= additiveExpression

additiveExpression:
        multiplicativeExpression
        | additiveExpression + multiplicativeExpression
        | additiveExpression - multiplicativeExpression

multiplicativeExpression:
        castExpression
        | multiplicativeExpression * castExpression
        | multiplicativeExpression / castExpression
        | multiplicativeExpression % castExpression

castExpression:
        unaryExpression
        | ( typeIdentifier ) castExpression

typeIdentifier:
        int | float | boolean

unaryExpression:
        postfixExpression
        | ^ unaryExpression
        | $ unaryExpression
        | ++ unaryExpression
        | -- unaryExpression
        | + unaryExpression
        | - unaryExpression
        | !  unaryExpression

postfixExpression:
        primaryExpression
        | postfixExpression [ additiveExpression ]
        | postfixExpression ([conditionalExpresssion (, conditioalExpression)*])
        | postfixExpression ++
        | postfixExpression --

primaryExpression:
        scopedId
        | literalValue
        | ( assignmentExpression )

literalValue:
        INT_LIT
        | FLOAT_LIT
        | STRING_LIT
        | true
        | false
```