NOKIA

Research Report

February 2003

Model Based Design of Communicating Systems

The methodology part developed by:

- Sari Leppänen	sari.leppanen@nokia.com
- Markku Turunen	markku.turunen@nokia.com

Appendix I: Backgrounder on UML 2.0 written by:

- Morgan Björkander	morgan.bjorkander@telelogic.com

This research report describes a model-driven design method for protocol engineering. It is a method that covers all phases from pre-standardization to final implementation. Modelling is service-oriented and based on the ideas of compositionality and externally observable behaviour, used also e.g. in [8][9], and refinement. In this paper, UML 2.0 is used as a modelling language. Many of the protocol specific basic concepts used in the methodology have already been described in [7].

The design process consists of four phases: Service Specification, Service Decomposition, Service Distribution and Implementation. Figure 1 roughly illustrates the design flow through the phases. The key idea is to derive the final implementation by refining step-by-step the executable service requirement model. Traceability between the models is supported by the well-defined workflow and tested with exhaustive simulation of the models. In the following paper, a detailed description will be given of each work phase with clarifying examples.

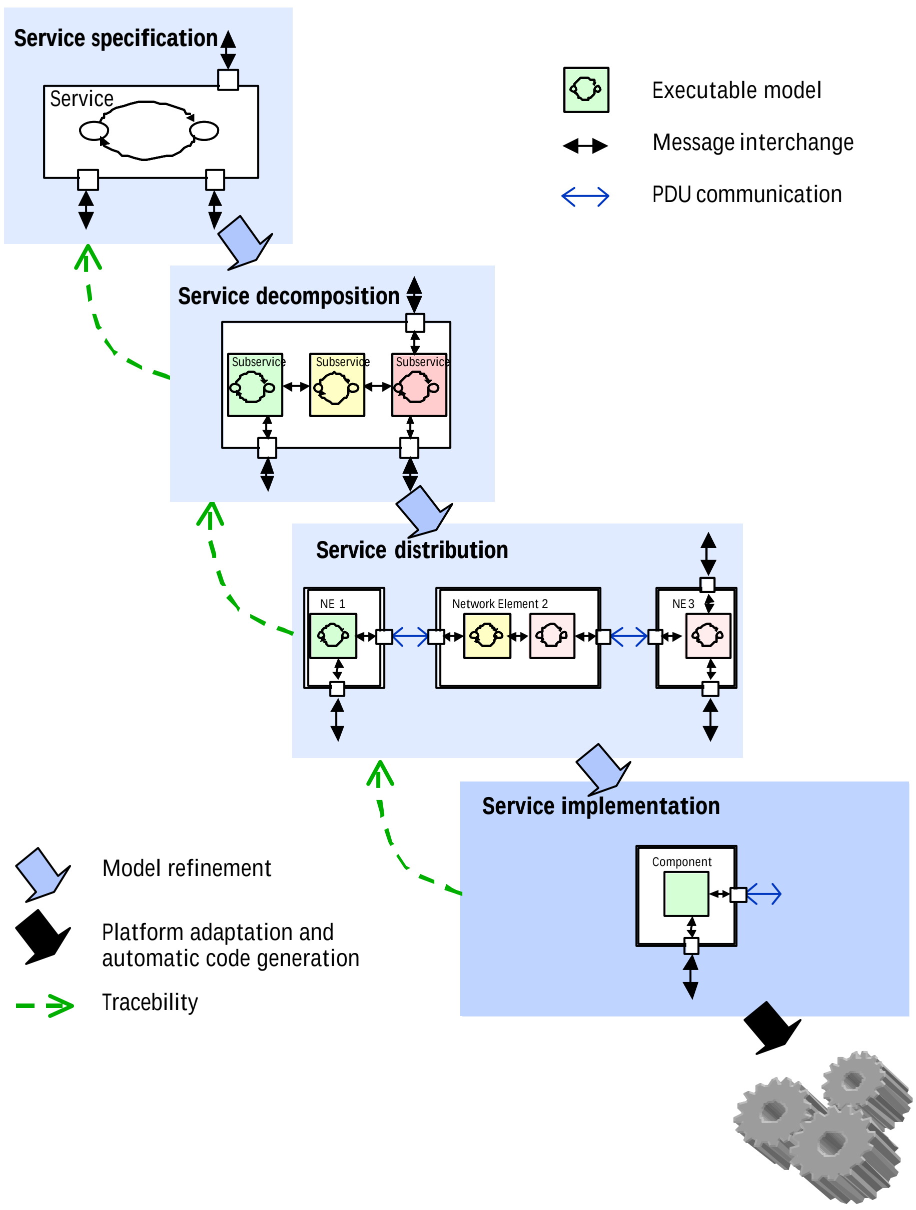


Figure 1: Model-driven design flow

The *Service Specification* phase describes the service provided by the system and the requirements set for the functionality. Functional requirements are modelled as a finite state machine, which also enables simulation. Non-functional requirements are documented separately. At this stage, the system implementing the service is regarded as a 'black box'.

The *Service Decomposition* phase provides a step-by-step breakdown of the service. From the top-down, it separates the service into smaller parts, i.e. *service components*. On various abstraction levels, decomposition produces several transparent views to the internal architecture of the system.

In the *Service Distribution* phase the service components are distributed over a given network architecture. The modularity of the model allows several distribution models and configurations of the system. Composition is used to encapsulate, from the bottom-up, the distributed functionality and, therefore, to define the modular internal architecture of the system.

Finally, in the *Implementation* phase the service component is integrated with the target platform and the code generated automatically.

Figure 2 roughly illustrates the idea of using decomposition and composition techniques in refining the service description to a system implementation. An extended, or 'stretched', design methodology covers all phases from service specification to a full implementation. Various groups of people participate in the different phases. For example, in protocol engineering the design methodology covers phases from pre-standardization to implementation. The Service Specification phase is related to stage 1 in the standardization process described in [4]. The Service Decomposition phase contributes mainly to standardization stage 2 and Service Distribution to stages 2 and 3.

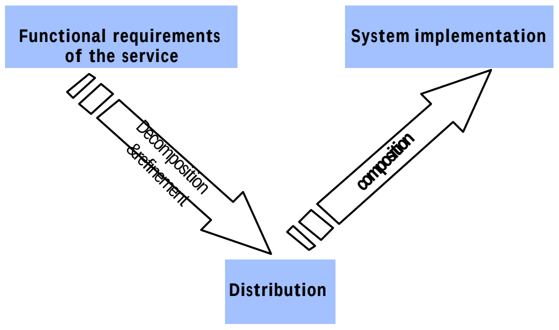

Functional requirements of the service
System implementation
Decomposition &refinement
composition
Distribution

Figure 2: Composition, decomposition and refinement in the design flow

The following paper will provide a more detailed description of the phases of this methodology. In each phase, the *class definitions, interface definitions, internal architecture description* and *behaviour specification* are described. Along with the methodology description, a case study, in which a *Third Generation Partnership Project (3GPP)* wireless communication protocol is modelled according to the main principles of our method, is used as an example. The protocol, *Position Calculation Application Part (PCAP),* is part of the *User Equipment (UE)* positioning system in the radio access network. PCAP is specified to manage the communication between the *Radio Network Controller (RNC)* and the *Stand-alone Assisted Global Positioning System Serving Mobile Location Center (*SAS) network elements (see Figure 3). The functional requirements for the RNC-SAS communication are specified in [1]. The PCAP specification [2] defines the *Iupc* interface and the corresponding signalling procedures, i.e. the functional description of the communication protocol.

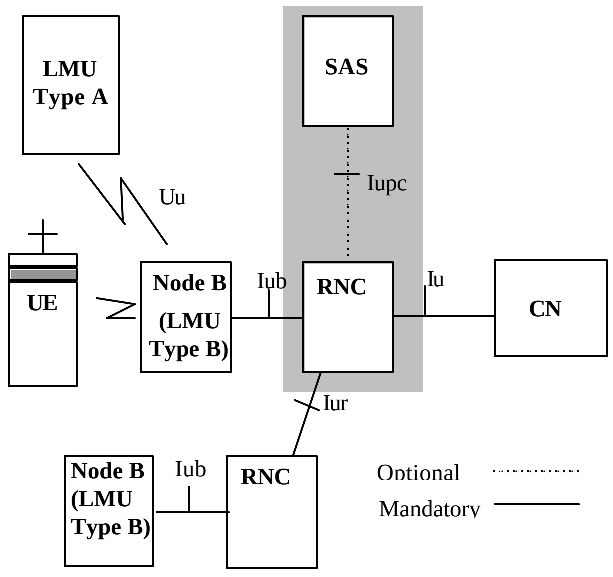
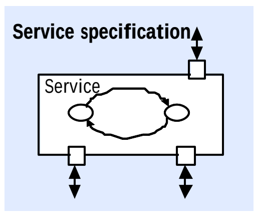


Figure 3: UMTS Network Architecture related to Position Calculation

1. SERVICE SPECIFICATION



The Service Specification phase specifies the functional requirements of the service with an executable state machine. The functional requirements are satisfied by a *valid externally observable behaviour* of the system. Signal exchange on various external interfaces of the system represents the externally observable behaviour of the system. By defining the correct execution order for these signal exchange actions, it's possible to obtain the specification for the valid externally observable behaviour of the system, which can be described with a finite state machine.

To be able to specify the behaviour it's necessary to define the external interfaces with sets of signals and signal parameters. To identify the external interfaces of the system a domain model for the system and its environment was created.

1.1 Concepts

In the Concepts definition work phase, the basic concepts used throughout the design process are identified and named. The paper takes the approach of a 'stretched' design process, which covers steps from the requirement specification to a full implementation and therefore involves people with very different backgrounds and perspectives. This illustrates the importance of concept definition as a cornerstone, upon which the rest of the work can be built.

Definitions include both the concepts specific for the design methodology and for the application area. Names and definitions for logical, structural and physical elements are fixed to increase the mutual understanding during the design process. In this model the application area is protocol engineering. The concepts are specified as UML stereotypes and are used later to clarify and increase the information content of various class diagrams.

«Network element»

A discrete telecommunications entity, which can be managed over a specific interface. Example: the Radio Network Controller (RNC) network element in a 3GPP system. [3]

«User»

An external entity that is not part of the system and uses the services provided by the system. Example: a person using a 3GPP system user equipment as a portable telephone. [3]

«Service»

A component of the portfolio of choices offered by service providers to a user, a functionality offered to a user. Example: positioning service, which can be used to locate a 3GPP system user equipment. [3]

«Service Component»

An independent piece of a service at any stage of decomposition. Components provide an abstraction and information hiding mechanism so that a component can be changed without requiring any changes to other components. Interaction between the modules is completely encapsulated by communication interfaces or (controlled) shared variables. Example: *PCService* component, part of the positioning service, encapsulates the interfaces and functionality concerning the requesting, reporting and termination of the position calculation operation.

1.2 Classes in Service Specification

This paper shows the modelling of entities that can receive asynchronous messages as *active classes*. Other concepts are modelled as *passive classes.* Entities, which are located outside of the modelled system, are specified as *external classes*.

In the Service Specification model the Service is specified as an active class. Users of the Service are specified as external classes. In addition, the used services (external capabilities utilized to produce the service) are specified as external classes. Internal components of the Service, which are not to be implemented (e.g. databases and algorithm libraries), are specified as external classes.

The defined classes are used to draw a domain model, where the existing information on the Service and its environment is described. The domain model is depicted with a class diagram, where the Service and relevant external entities appear as classes or stereotypes. The relationships between them are described using *associations*.

Figure 4 illustrates the domain model of *PCAPService*.

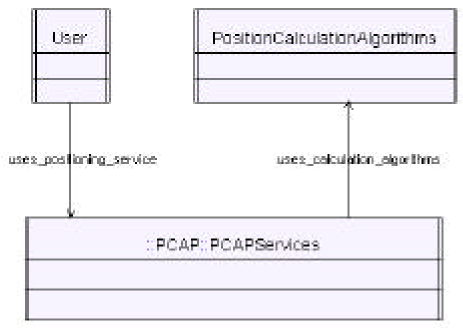


Figure 4: Domain model for the PCAP service

1.3 Interfaces in Service Specification

According to the UML definition, an interface is a structured classifier, which may not be instantiated. Instead, it is used for grouping a set of signals, which are used for communication by the class that implements the interface. A class that implements an interface is said to *realize* that interface, and therefore able to receive the signals declared in the interface. A class can also *require* interfaces enabling it to send signals to other active classes realizing the corresponding interfaces.

A *Service Interface* is a communication point for bi-directional signal exchange between the system and its environment. The concept Service Interface contains both the required interfaces, where the system provides its services to the *User Interfaces (Users)* and the realized interfaces, where the system uses other services provided by external entities *(Used Service Interfaces)*.

Service Interfaces are used by the system providing the service for communication with the environment. Service Interfaces are *external interfaces* of the system. Through the signal exchange on the external interfaces part of the behaviour of the system becomes visible. This behaviour is called the *externally observable behaviour* of the system.

During the Service Specification phase the actual interface definitions, i.e. signals and signal parameters, are first given for the Service Interfaces. To link the Service interfaces to the classes in the domain model, *ports* corresponding to the interface definitions in the classes are specified. *Ports* are named interaction points of an active class. They specify the realized interface of the class and the required interfaces from other classes.

In the PCAP example, the external interfaces for the *PCAPServices* class are specified according to the information in the *Domain Model* diagram. Signals used in communication in the User Interface are grouped in the interface definitions *I_UserToPCAP* and *I_PCAPToUser*. Sets of signals grouped under the interface definitions *I_AlgorithmToPCAP* and *I_PCAPToAlgorithm* are used for communication on the Service Interface with the external entity containing the computation logic and algorithms for actual position calculation. The User Interface is instantiated as *user_port* and the Service Interface towards the position calculation entity as *calculation_port*.

1.4 Architecture in Service Specification

An architecture diagram defines the internal run-time structure of an active class in terms of other active classes. The architecture diagram specifies how UML objects are instantiated and used together to form a system or a part of a system.

The Service class specified in the Service Specification phase does not have an internal structure. In order to define the service requirements on this level of abstraction it's desirable to observe the system providing the service as a 'black box' and hide all implementation specific details, including the internal structure of the system.

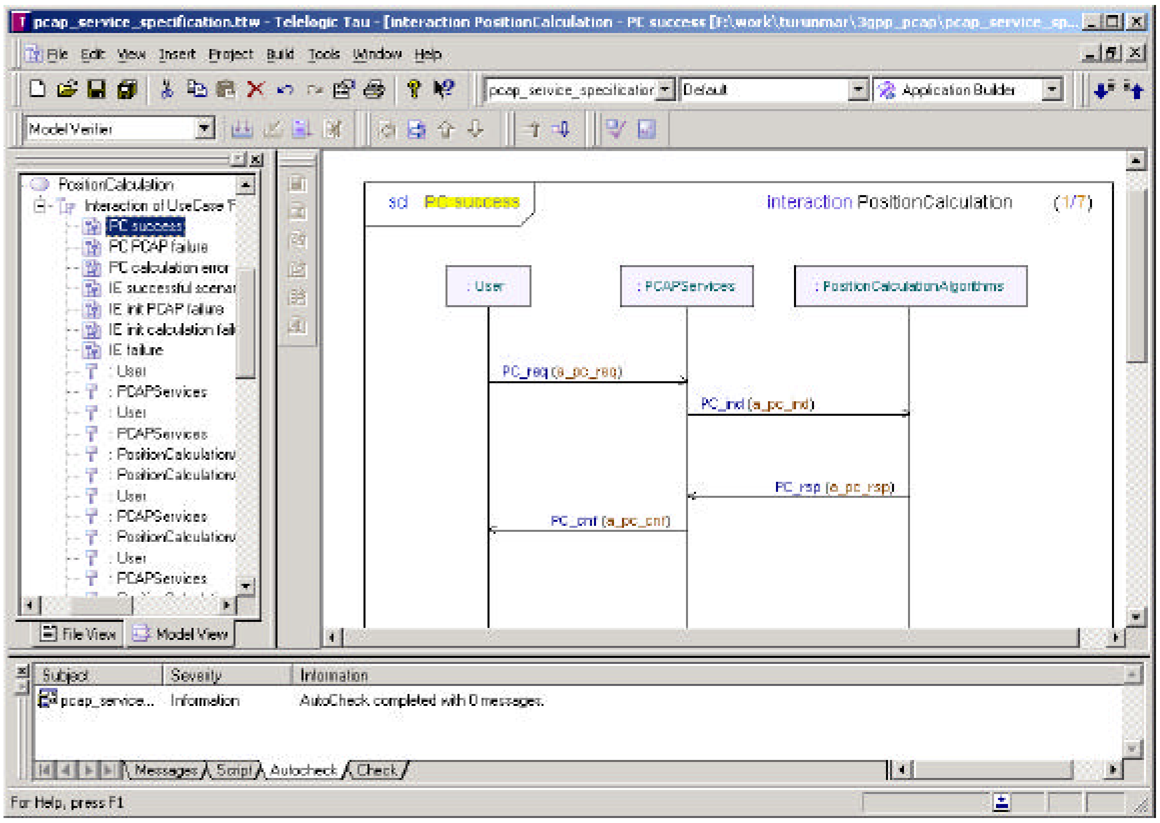
The domain model can be considered an architecture description of the Service Specification phase. It describes the environment of the service, and thereby the structure of the service context. The domain model can be described either with a class diagram or with an architecture diagram. The domain model of the PCAP example is described as a class diagram in Figure 4.

1.5 Behaviour in Service Specification

Communication on the external interfaces represents the externally observable behaviour of the system. The valid externally observable behaviour on the User Interfaces of the system providing the service satisfies the functional requirements set for the service. By specifying the valid externally observable behaviour as a finite state machine it's possible to achieve an *executable requirement specification* for the service. Communication with the entities providing the used services and the other external entities is also described in the specification.

The preceding work phase already defined the signals for the Service Interfaces. By defining now the correct execution order of the signals it's possible to obtain the description of the valid behaviour for each Service Interface. Sequence diagrams can be used to sketch the signal exchanges for various *use cases*; both for the normal cases and for various error cases.

In Figure 5, the Model view on the left shows Sequence diagrams for the normal case, *PC success*, and for two error cases. *PC PCAP error* describes an error case due to a protocol error and *PC calculation error* describes a scenario where an error occurs in the actual position calculation functionality, external to the system.



Subject	Severity	Information
pcap_service...	Information	AutoCheck completed with 0 messages.

Figure 5: Scenario in case of successful Position Calculation

Based on the sequence diagrams it's possible to formulate one or several state machines to describe the behaviour on the Service interfaces. We can, for example, specify in separate state machines part of the requirements concerning signalling in one use case on one interface. For example, Figure 5 could be used as basis to describe two state machines: the functionality of the *PC success* use case on both Service Interfaces. Figure 6 presents part of the Service

Specification state machine for the User Interface, instantiated with the *user_port* (see chapter 1.3). The 'big' state machine, describing the service requirements as a whole, can be composed from these 'small' ones.

Due to the high abstraction level of the model hiding all implementation specific details, nondeterminism is quite inevitable in the model. Nondeterministic choices can be resolved with interactive simulation or, in case of automatic execution of the model, with constants or variables together with randomised value generators. Figure 6 illustrates an example of using non-determinism in a state machine.

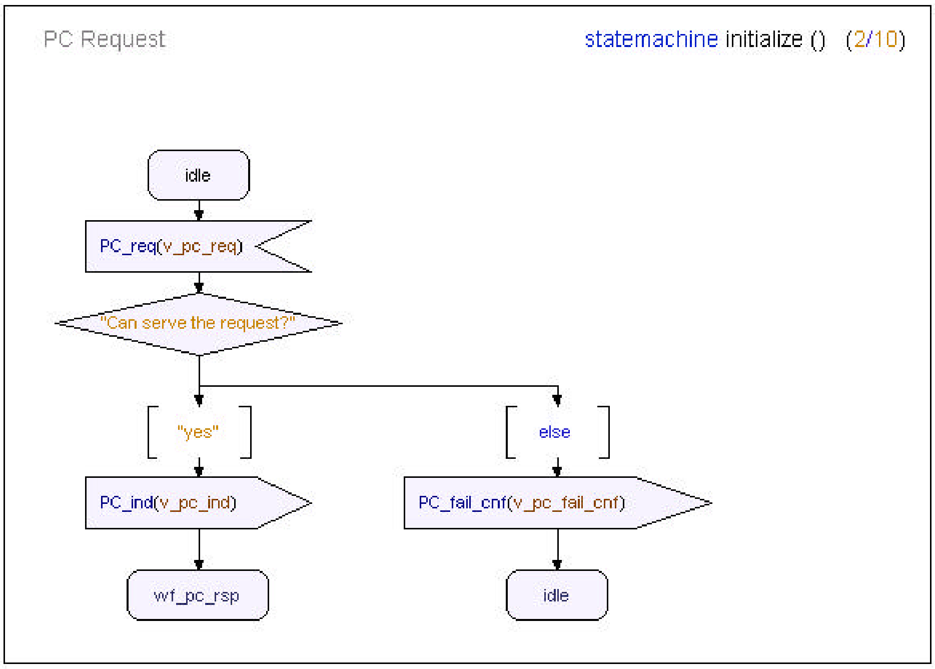


Figure 6: A non-deterministic behaviour in Service Specification

1.6 Executable specification in Service Specification

A remarkable strength of our design methodology is the executability of the models. The ability to execute the models on various abstraction levels enables, for example, testing of new ideas and proposals within the standardization work. Executability enables also validation and evaluation of the core functionality.

For simulation and testing purposes an appropriate configuration of the system is defined. From the ports defined for the Service class we choose the ones we want to observe and include them into our configuration. Simulation can be run either interactively or automatically. For automatic simulation, e.g. for verification purposes, separate *user* and *monitoring processes* can be defined and connected to the instantiated ports of the system. Automatic handling of test inputs and outputs enables more exhaustive simulation, increasing the confidence in the reliability of the system.

Figure 7 presents the configuration for the interactive simulation of PCAP Service with both Service Interfaces.

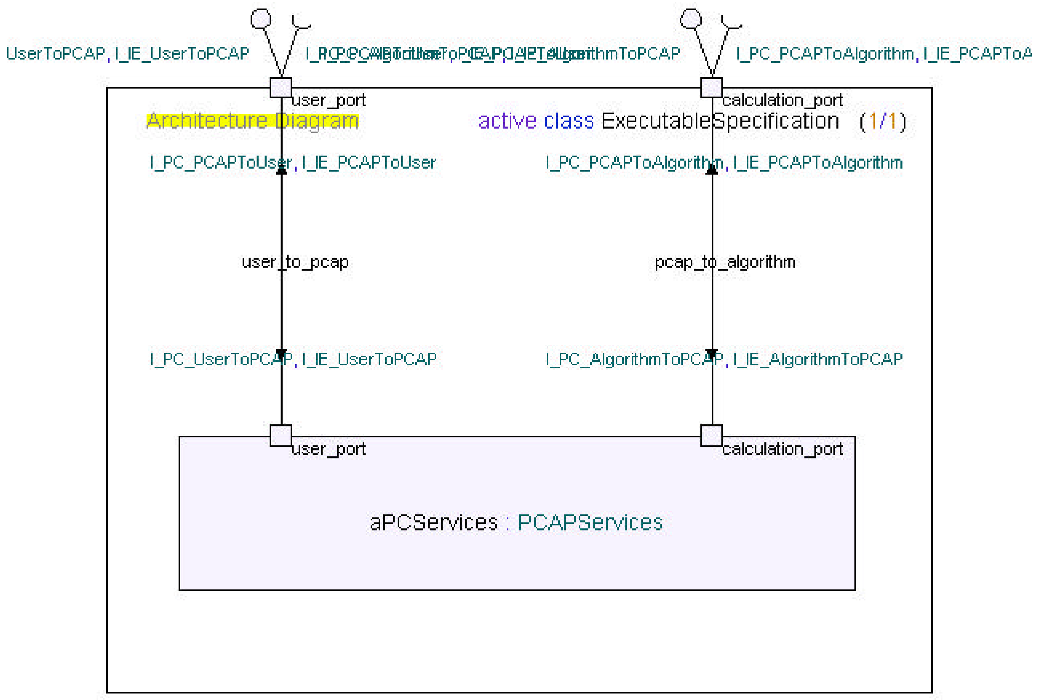
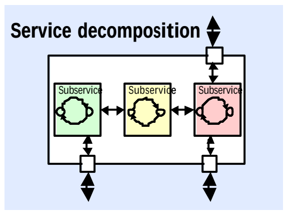

UserToPCAP, I_IE_UserToPCAP
I_PC_PCAPToUser, I_IE_PCAPToUser
I_PC_AlgorithmToPCAP, I_IE_AlgorithmToPCAP
I_PC_PCAPToAlgorithm, I_IE_PCAPToA
user_port
calculation_port
Architecture Diagram
active class ExecutableSpecification (1/1)
I_PC_PCAPToUser, I_IE_PCAPToUser
I_PC_PCAPToAlgorithm, I_IE_PCAPToAlgorithm
user_to_pcap
pcap_to_algorithm
I_PC_UserToPCAP, I_IE_UserToPCAP
I_PC_AlgorithmToPCAP, I_IE_AlgorithmToPCAP
user_port
calculation_port
aPCServices : PCAPServices

Figure 7: Configuration of executable Service Specification

2. SERVICE DECOMPOSITION



In the Service Decomposition phase the logical functionality implementing the required service is recursively decomposed into smaller *service components*. Decomposition proceeds in top-down manner. Each service component is characterized and notes regarding rational distribution (e.g. according to the relationships to other services) are made on them.

The Service Decomposition model represents several views of the internal architecture of the system on various abstraction levels. The ultimate aim is to produce a set of smaller models – service components – each of which encapsulates a logically consistent functional unit. Decomposition enables the usage of different distribution models for various network architectures. Additionally, decomposition enables the flexible definition of several system configurations and, providing a well-defined modularisation of the system is to be implemented, also efficient planning for resource allocation.

2.1 Classes in Service Decomposition

In the Service Decomposition phase classes are defined for the service components arising from the decomposition of the service. Later, port definitions are attached according to the interfaces defined for the service component and the state machine describing the externally observable behaviour of the service component to these classes.

Decomposition is a recursive procedure starting from the service class specified in the Service Specification model. The decomposition procedure results in a *decomposition tree*, where each node corresponds to a service component, a logically consistent functional unit. Service components are represented by classes to encapsulate the definitions of external interfaces and externally observable behaviour of a service component. The root of the decomposition tree is the service class specified in the Service Specification model. When the node on depth *n* represents a black-box view of a service module, the transparent view of the same element is represented by a set of nodes on depth *n+1*. The leaves of the decomposition tree are the smallest functional elements (e.g. procedures), which can, within reason, be encapsulated as separate units.

The *PCAPServices* class is the root of the decomposition tree. In the Service Specification phase the external interfaces and the behaviour are defined for the *PCAPService* class. At this stage, the *PCAPService* class is decomposed into two functional entities and creates the corresponding class definitions. The *PCService* class contains the functionality related to the actual position calculation process. The *IEService* contains the functionality related to the transmission of additional information needed in the actual position calculation process.

2.2 Interfaces in Service Decomposition

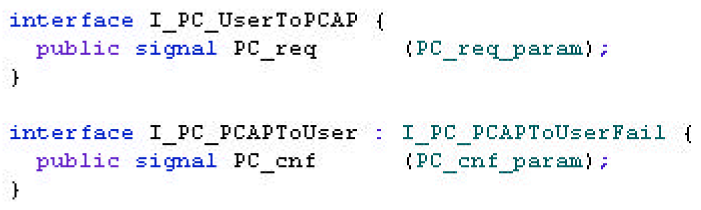
The external interfaces of the service components are formed and specified according to the functional service decomposition. This usually means no introduction of new definitions, but mainly the regrouping of the existing signal definitions.

A step in the decomposition process reveals the internal structure of a service component. A step in the decomposition process also divides the external interfaces into several parts and turns internal, invisible interfaces of a service component into external, visible interfaces of the contained service components. In the decomposition tree part of the internal, non -visible interfaces on depth *n* are external, visible interfaces on depth *n+1*.

In the Service Decomposition phase all the *internal interfaces* of the system providing the service are defined. An internal interface is not visible as a port instance in the class symbol of a service component. Instead, in the architecture diagram, which describes the internal structure of the service component, the internal interfaces are visible through port instantiations of parts and connectors between the ports. The external interfaces of the service component appear as port instances on the outer borders of the architecture diagram and they are connected to the internal interfaces.

When a service component is decomposed into smaller service components and a class is defined for each component, the external interfaces begin to be decomposed. Within the port definitions of the smaller service components is specified which part of the external interface of the original service component is implemented by the service component concerned. Next the internal structure of the decomposed service component is defined in an architecture diagram. If there is need for interaction between the internal components, the corresponding internal interfaces have to be defined at this stage. An elementary requirement for compositionality and for 'pure' modularity is the clear encapsulation of interactions between the components. Interactions between the components here are allowed only through signal exchange.

In the PCAP example the *PCService* component and the *IEService* component do not have mutual communication, so the decomposition does not create the need for new internal interface definitions. The external interfaces specified in the Service Specification phase are divided according to the functional decomposition (see Figure 9). Figure 8 describes the *PCService* specific part of the User Interface. The *user_port* and *calculation_port* of *PCAPService* is also decomposed to the *PC* specific part and *IE* specific part correspondingly and attached to the *PCService* class and *IEService* class.



```
interface I_PC_UserToPCAP {
  public signal PC_req       (PC_req_param);
}

interface I_PC_PCAPToUser : I_PC_PCAPToUserFail {
  public signal PC_cnf       (PC_cnf_param);
}
```

Figure 8: User Interface definition for *PCService* component

2.3 Architecture in Service Decomposition

An architecture diagram defines the internal structure of a decomposed service component in terms of other service components. In the Service Decomposition phase several architecture diagrams are usually produced, at least one for each decomposition level.

The architecture diagram instantiates as *parts* the classes defined for the smaller service components composing the original service component. To each part ports are attached realizing and requiring the signals defined for both external (part-to-environment) and internal (part-to-part) communication.

Interfaces are visible through ports defined for the parts and the *connectors* between the ports. A connector specifies a medium that enables internal and external communication. Connectors can visualize communication paths intuitively, but may also be omitted. The mandatory requirement for establishing the communication is that the *required - realizes* definitions in the port definitions match consistently and completely inside an architecture diagram. A connector may be uni- or bidirectional and specifies the allowed signals for each direction. When the number of signals is large, it is more convenient to refer to an interface or define a signal list for each direction of the connector.

The external interfaces of the decomposed service component appear as ports on the outer borders of the architecture diagram. Each of the external ports has to be connected to one or several internal ports. In case there are several internal ports corresponding to one external port, the composite signal set of the internal ports has to be equivalent to the set of signals defined for the external port. Ports related to internal communication are connected correspondingly.

Figure 9 presents the internal architecture of the *PCAPService* consisting of the parts instantiating the classes *PCService* and *IEService.*

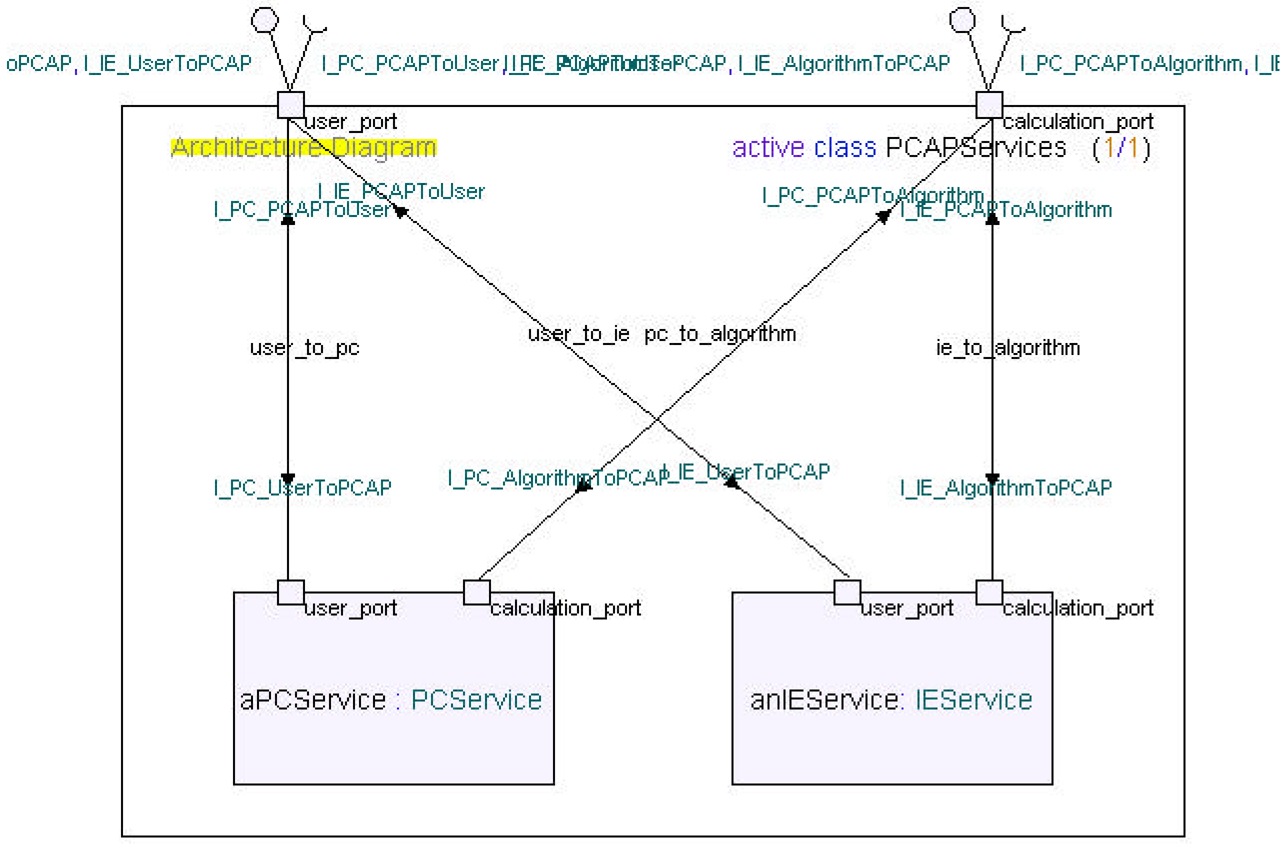


Figure 9: internal architecture of the PCAPService

2.4 Behaviour in Service Decomposition

A step in the decomposition process turns part of the internal interfaces of a service component into external interfaces of the contained service components. Therefore part of the internal behaviour of the decomposed service component turns into externally observable behaviour of the contained service components.

The main principle of this design methodology is that in the modelling phase only the externally observable behaviour is specified. The specification is refined, i.e. more of the 'total' behaviour is specified, as the decomposition proceeds revealing new internal communication interfaces, and thereby creates the need for new behaviour specifications. The specification of the behaviour means essentially the definition of the order of exchanged signals, described in a state machine.

In the Service Decomposition phase the behaviours of all service components are specified. In a decomposition step at least two new service components come into existence. First the behaviour is divided, i.e. the functionality, specified in the earlier phase for the decomposed service component according to the regrouping of the external signalling (interface definitions) and internal architecture definition (architecture definition). If the new service components arising from the decomposition have mutual communication and therefore new internal interfaces are specified, then the behaviour on these interfaces is defined and integrated to the other, existing behaviour specified earlier. In the same way as the behaviour was specified in the Service Specification phase, sequence diagrams can be used here in drafting the externally observable behaviour on separate interfaces of the service component. The behaviour of a service component can be specified either as many separate state machines, e.g. one for each interface, or as one extensive state machine integrating externally observable behaviour on all interfaces of one service component.

In Figure 10 the sequence diagram for successful execution sequence of an IE procedure is presented. Repetition occurring as a self-loop in the final state machine is highlighted.

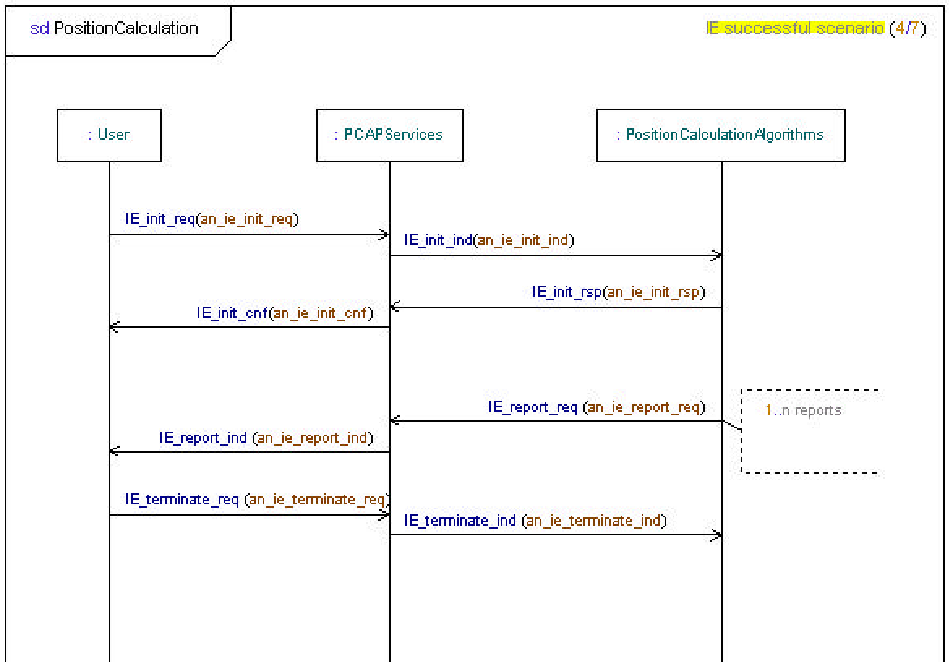
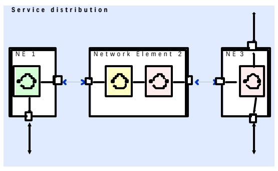

sd PositionCalculation
IE successful scenario (4/7)
: User
: PCAPServices
: PositionCalculationAlgorithms
IE_init_req(an_ie_init_req)
IE_init_ind(an_ie_init_ind)
IE_init_rsp(an_ie_init_rsp)
IE_init_cnf(an_ie_init_cnf)
IE_report_req (an_ie_report_req)
1..n reports
IE_report_ind (an_ie_report_ind)
IE_terminate_req (an_ie_terminate_req)
IE_terminate_ind (an_ie_terminate_ind)

Figure 10: Scenario in case of successful Information Exchange

2.5 Executable specification in Service Decomposition

For testing, separate simulation configurations, 'test benches', are designed for each service component. Once there is confidence in the correct functionality of a service component as a separate module in isolation, a new simulation configuration can be defined for integrating several service components to test their interactions and parallel functionality.

3. SERVICE DISTRIBUTION



In the Service Distribution phase, the functionality of the system is adjusted to a given physical structure of a network. The distribution is based on the information of the network architecture and the service component produced in the Service Decomposition phase. According to the network architecture information the service components are located in various network elements and protocol layers. Decomposition, and thereby the modularity, supports different distribution models for various network architectures and also flexible configuration of the system.

Structural elements, such as protocols, subprotocols and *computational contexts*, are specified by grouping the service components according to their location and functionality. In other words, the composition of service components in a bottom-up manner produces the structural elements.

The *PDU interfaces* between communicating structural elements located in separate network elements are defined. New PDU interfaces bring forth new externally observable behaviour, which has to be defined and merged with the other *behaviour of the structural elements*.

3.1 Classes in Service Distribution

In the Service Decomposition phase the service components are decomposed into smaller service components in top-down manner. The aim is to define small but still logically consistent functional units with well-defined interfaces.

In the Service Distribution phase the previously defined service components are distributed over several physical network elements. In distribution, both the network architecture description and the Service Decomposition model are taken into account. The three main cases for distribution are as follows.

The first and the most trivial case of distribution is that separate service components are located in different network elements. If there is no communication between the distributed service components, the distribution requires no further actions or definitions. A class is specified for each distributed service component in the Service Distribution model. No changes for the interface definitions are required.

In the second case the separate service components communicate with each other and the signal exchange is encapsulated by the internal interfaces. In the distribution the interfaces they use for internal communication are 'opened up' as a *PDU interface*. For example, in Figure 11 service components *SC1* and *SC2* have interaction and, after distribution, the internal interface between them is opened up as a PDU interface. Also in this case, a class is specified for each distributed service component to the Service Distribution model. Changes to the interface definitions are described later in the interface definition work phase.

In the third and the most complicated case of distribution, a single service component is distributed over several network elements. The external interfaces, i.e. the Service Interfaces, of the service component are distributed according to the distribution of the functionality. A class is specified for each *peer entity* (a part of the distributed service component) in different network elements. New PDU interfaces are defined between the peer entities to enable interaction and peer-to-peer communication between the distributed parts of a service component. For example, in Figure 11 the functionality of the service component *SC3* is distributed over the network elements NE2 and NE3. Classes are specified for the peer entities *SC3a* and *SC3b*. A PDU interface is defined between the peer entities to enable interaction and thereby also the preservation of the total functionality of the distributed service component on the system level.

The Service Distribution phase is the most challenging phase in the design flow and should be done with extra care and precision. Distribution is always case sensitive. Besides the main cases presented here, there are definitely also other cases. For example, a service component may be replicated in several network elements.

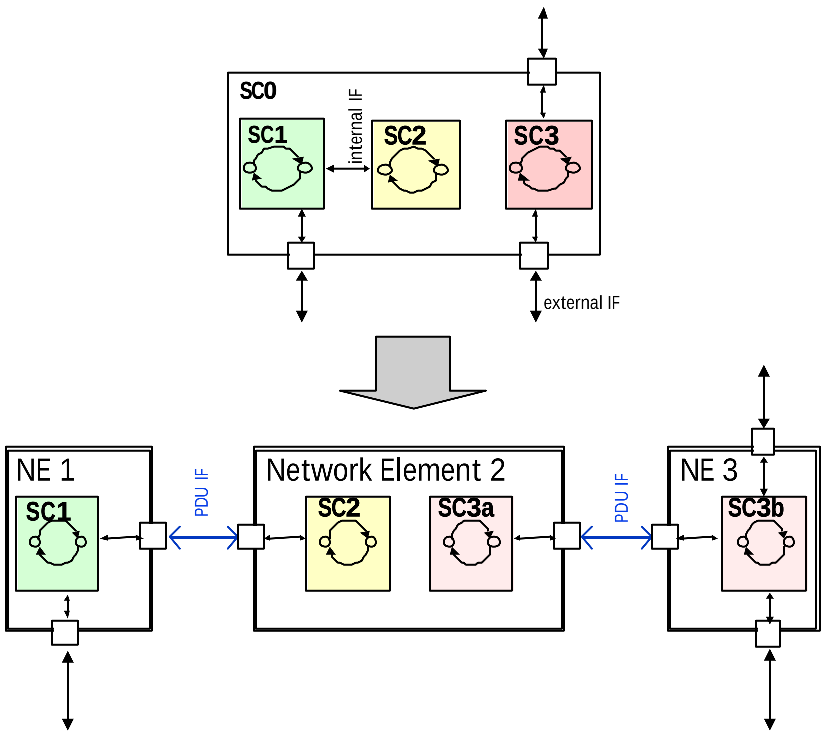


Figure 11: Service Distribution

Grouping, or composition, of the distributed service components in a bottom-up manner produces *structural elements*, which are also defined as classes in the Service Distribution model. Composition, like the decomposition in the reversed direction, produces structural components on several abstraction levels. For example, if two service components inside one network element are in tight interaction and their procedures may even be executed in an interleaved manner, it is reasonable to merge these two into one *computational context*. Furthermore, all computational contexts participating in interactions between two network elements can be encapsulated into a *subprotocol*. A *protocol* consists of several subprotocols, and a *layer* consists of several protocols. The ultimate result will be a modular, well-defined description for the internal architecture of the distributed service.

PCService and *IEService* components are distributed over the network architecture described in Figure 3. These service components are executed independently of each other, so they are specified as their own computational contexts. The computational context is always specified inside a network element, so the network element specific computational contexts of this system are named as *SAS_PCService*, *RNC_PCService*, *SAS_IEService* and *RNC_IEService*. The subprotocols of the PCAP protocol, *SAS_PCAP* and *RNC_PCAP,* encapsulate all the computational contexts inside one network element.

3.2 Interfaces in Service Distribution

In the Service Distribution phase the functionality encapsulated by the service components is distributed. The distribution causes some changes to the interface definitions, mainly concerning the interfaces used for communication between the service components.

In the simpler cases of distribution, separate service components are located to different network elements. If there is no communication between the distributed service components, the distribution does not require any changes to the interface definitions. In case there is communication between the distributed service components and the signal exchange is encapsulated by the internal interfaces, the internal interface is 'opened up', or made visible on the system level, as a *PDU interface*. PDU interface is an interface used for communication between service components in separate network elements.

In the third case of distribution, a single service component is distributed over several network elements. The external Service Interfaces of the service component are distributed according to the distribution of the functionality. The distribution of the functionality produces peer entities located in separate network elements. To enable the necessary interaction by means of communication, it's necessary to define new PDU interfaces between the peer entities. The signals for the PDU interfaces come from the behaviour specification. The PDU interfaces are grouped, or composed together, according to the composition of the structural elements so, that finally there is one PDU interface for one protocol.

Figure 12 illustrates the interfaces of the distributed *PCService* component on the RNC network element side. The part of the User Interface, which is specific to the PCService component, is defined in the interfaces *I_PC_UserToPCAP* and *I_PC_PCAPToUser. I_PC_SASToRNC* and *I_PC_RNCToSAS* interfaces define the PDU communication of *PCService.* When *I_PC_SASToRNC* and *I_PC_RNCToSAS* interfaces are composed together with *I_IE_SASToRNC* and *I_IE_RNCToSAS*, the corresponding PDU interfaces of the *IEService*, this results in the *I_SASToRNC* and *I_RNCToSAS* interfaces, which define the PDU communication of the PCAP protocol.

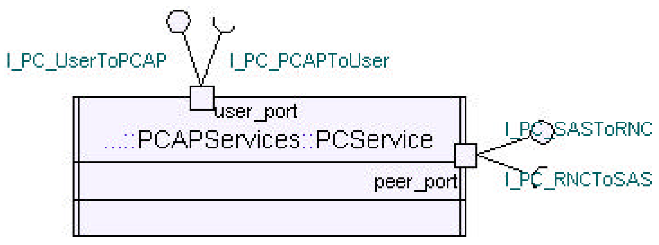


Figure 12: Interfaces of the distributed PCService

3.3 Architecture in Service Distribution

In the Service Decomposition model, the internal structures of the Service Components are described on various abstraction levels with architecture diagrams. In the Service Distribution model architecture diagrams are used to describe the internal structure of structural elements, i.e. subprotocols, protocols and layers, on various abstraction levels. In the architecture diagram the classes defining the smaller structural components composing the structural component concerned are instantiated as parts. Ports attach the interfaces specified earlier to the parts. Connectors can be used to visualize the point-to-point communication. As in the earlier phases, ports encapsulating the external interfaces of the structural element concerned have to be connected to the internal ports of the contained structural elements.

Figure 13 describes the internal structure of the PCAP protocol on the RNC side. The PCAP protocol consists of two computational contexts: *PCService* and *IEService*. In the PCAP example, the computational contexts could be called subprotocols as well, since there is no further structurisation. The *user_port* of the PCAP protocol, i.e. the class *PCAPServices_RNC*, is connected to the *user_ports* of the *PCService* and *IEService* encapsulating the computational context specific interfaces. Similarly, the *peer_port* of the PCAP protocol encapsulating the PDU interfaces is connected to the corresponding ports of the contained computational contexts.

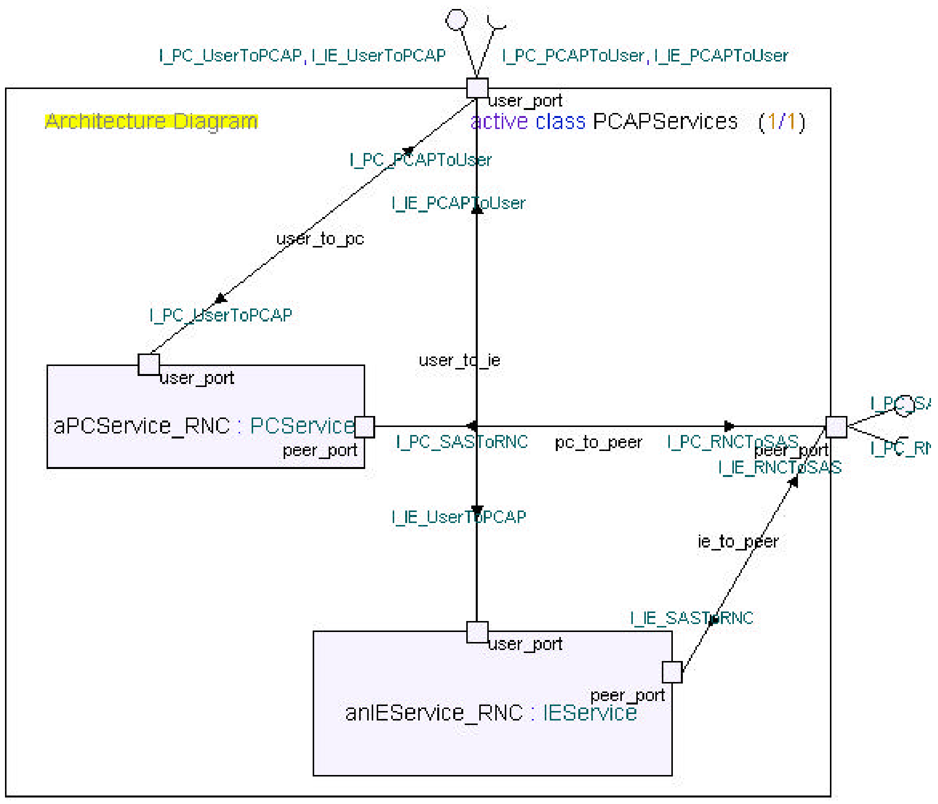


Figure 13: Internal structure of the PCAP protocol, RNC side

3.4 Behaviour in Service Distribution

In the Service Distribution phase, some of the behaviour defined in the Service Decomposition phase is split into several interacting parts. In the preceding class definition work phase, classes were specified for each computational context. Now, in the behaviour definition work phase, we describe the functionality of each computational context class with state machines. Since the corresponding computational contexts in different network elements use PDU communication to interact with each other, the state machines defined here are peer state machines with respect to the PDU communication. This means that the sending of a PDU in one state machine must have corresponding receiving action in its peer state machine.

A computational context handles two different kind of communication, namely the ‘horizontal’ PDU communication and the ‘vertical’, in-stack communication with the users. The horizontal PDU communication is specified in the protocol standards to guarantee the interoperability of network elements of different manufacturers. The peer state machines for the PDU communication reside inside the same protocol, so they belong to the same specification domain. The vertical, in-stack communication is implementation-specific. The corresponding state machines for the vertical communication reside outside of the protocol entity. This split-level communication is presented with hierarchical state machines.

A statechart diagram visualizes a state machine. There are two different styles of drawing statechart diagrams supported in Tau/Developer. The state-oriented view of a state machine gives a good overview of a complex state machine but is less practical when each transition contains more behaviour and when the behaviour must be described more explicitly (i.e. making a design in a state machine). For this reason, it is also possible to describe the state machine in a transition-oriented way, with explicit symbols for different actions that can be performed during the transition. These two styles are combined in the hierarchical state machine. The behaviour related to the horizontal PDU communication on the main level of the hierarchical state machine is described with the state-oriented way. All the states on the main level are *composite states*, i.e. states composed by other states and transitions. The states are named according to the procedures defined for the computational context. Behind each composite state there is another state machine describing the 'vertical' communication with the user. This state machine is described in the transition-oriented way, since besides the stack-internal communication also all internal computation is included in this state machine.

In the horizontal state machine, all transitions from one state to another are either reception of PDUs or sending of PDUs. In the vertical state machine, all transitions are correspondingly related to *service primitives*. The control switch between the horizontal and vertical state machines is handled with *named entry and exit connection points*. An entry connection point is a named starting point for entering a composite state and an exit connection point is a named exit point for leaving a composite state. Reception of a PDU in the horizontal state machine causes a control switch to the vertical state machine through a named entry connection point. Sending of a PDU in the horizontal state machine is triggered by a named exit from the current composite state of the vertical state machine. If the triggering event is specified for the composite state in the horizontal state machine then the transition will cause an exit of the composite state (and substates) for a new state in the horizontal state machine. This feature enables a flexible and easy way of defining the reception of special signals, e.g. exception or termination, in any state or in a certain set of states.

In the PCAP example, the main level of the hierarchical state machine describes the behaviour of the *IEService* computational context (see Figure 14). On this main level state machine, the set of transitions consists of receptions and sendings of PDUs defined in the *IEService* PDU interfaces. Figure 15 illustrates one state diagram for the vertical state machine behind the *idle* composite state.

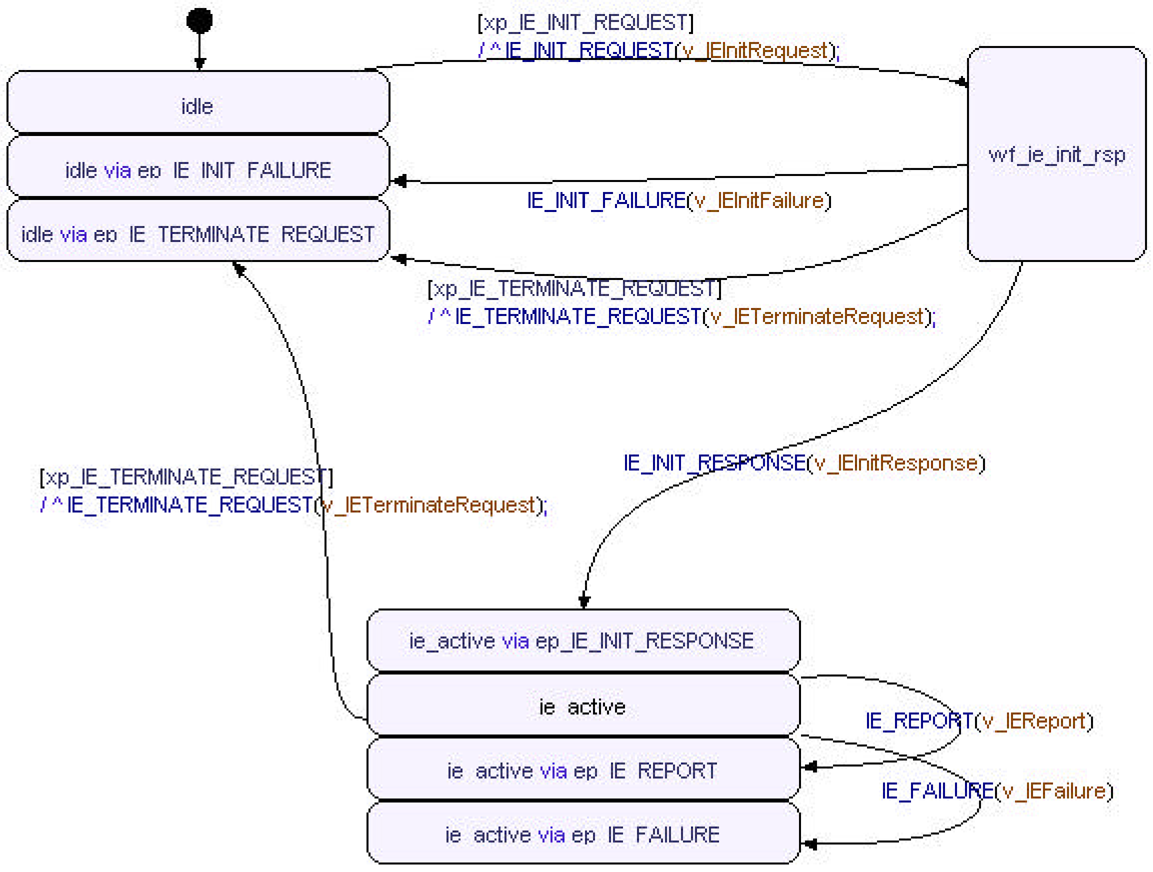

[xp_IE_INIT_REQUEST]
/ ^IE_INIT_REQUEST(v_IEInitRequest);
idle
idle via ep IE INIT FAILURE
idle via ep IE TERMINATE REQUEST
wf_ie_init_rsp
IE_INIT_FAILURE(v_IEInitFailure)
[xp_IE_TERMINATE_REQUEST]
/ ^IE_TERMINATE_REQUEST(v_IETerminateRequest);
[xp_IE_TERMINATE_REQUEST]
/ ^IE_TERMINATE_REQUEST(v_IETerminateRequest);
IE_INIT_RESPONSE(v_IEInitResponse)
ie_active via ep_IE_INIT_RESPONSE
ie active
ie active via ep IE REPORT
ie active via ep IE FAILURE
IE_REPORT(v_IEReport)
IE_FAILURE(v_IEFailure)

Figure 14: PDU state machine of the IEService class

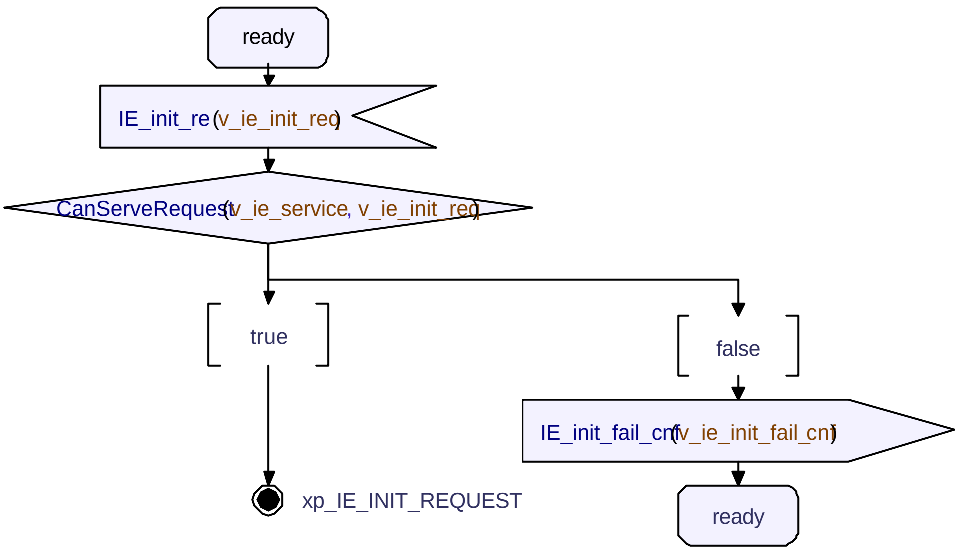

ready
IE_init_re(v_ie_init_req)
CanServeRequest(v_ie_service, v_ie_init_req)
true
false
IE_init_fail_cnf(v_ie_init_fail_cnf)
xp_IE_INIT_REQUEST
ready

Figure 15: Inside the 'idle' state

3.5 Executable specification in Service Distribution

With various configurations, it's possible to simulate and test either the whole protocol implementation or the protocol peer entities separately. Additionally, simulation and testing can be performed observing only the chosen interfaces.

Figure 16 presents the configuration for simulating the PCAP protocol, implementing the Iupc interface defined in [2]. In the simulation, it's desirable to observe the user interfaces and therefore hide the PDU interface between the peer entities.

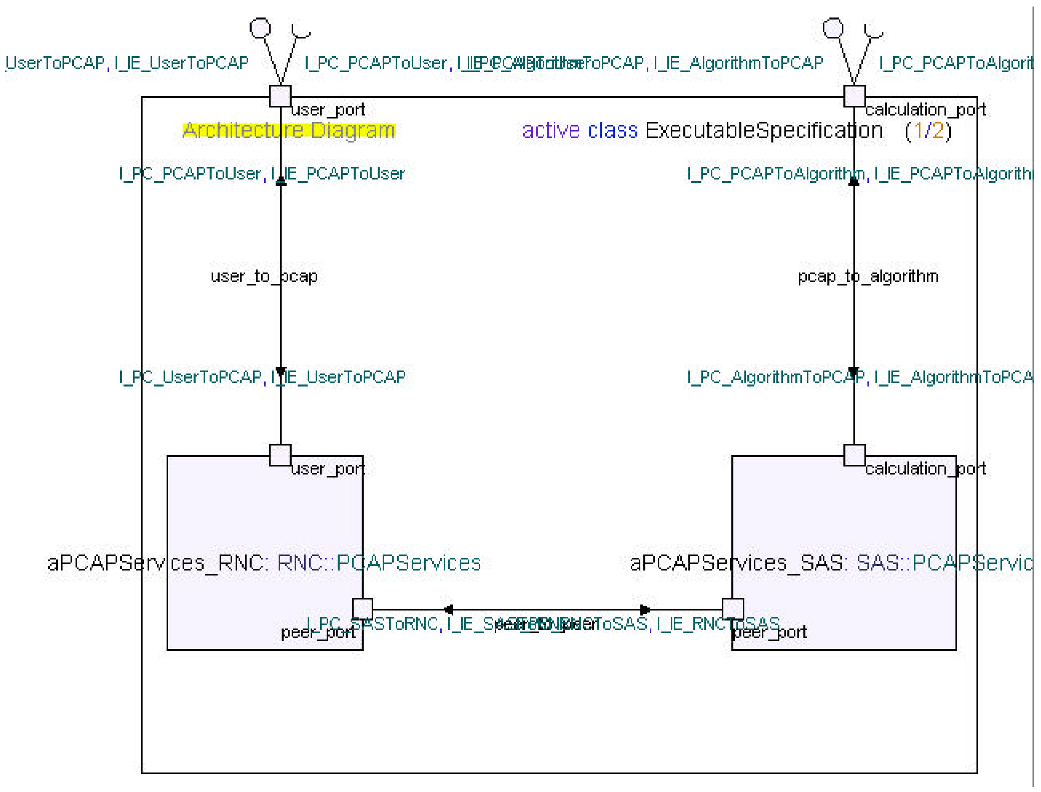
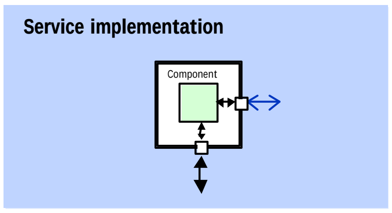

UserToPCAP, I_IE_UserToPCAP
I_PC_PCAPToUser, I_IE_PCAPToUser
I_PC_AlgorithmToPCAP, I_IE_AlgorithmToPCAP
I_PC_PCAPToAlgorit
user_port
calculation_port
Architecture Diagram
active class ExecutableSpecification (1/2)
I_PC_PCAPToUser, I_IE_PCAPToUser
I_PC_PCAPToAlgorithm, I_IE_PCAPToAlgorith
user_to_pcap
pcap_to_algorithm
I_PC_UserToPCAP, I_IE_UserToPCAP
I_PC_AlgorithmToPCAP, I_IE_AlgorithmToPCA
user_port
calculation_port
aPCAPServices_RNC: RNC::PCAPServices
aPCAPServices_SAS: SAS::PCAPServic
I_PC_SASToRNC, I_IE_SASToRNC
I_PC_RNCToSAS, I_IE_RNCToSAS
peer_port
peer_port

Figure 16: Configuration of executable Service Distribution

4. IMPLEMENTATION



The goal of the Implementation phase is to produce the Implementation model, from which code can be automatically generated for the target platform. The phase can be divided into two subphases, a generic one and a target platform specific one. In the generic implementation subphase the Service distribution model is enriched with the missing functional features. The added features include support for several parallel communication sessions for computational contexts, realization of the virtual PDU communication between the peer entities in different network elements and error handling. In the target platform specific subphase target platform and configuration issues are taken into account. The following sections illustrate actions during the generic subphase.

4.1 Classes in Implementation

The active computational context classes, specified in the Service Distribution phase, are used as basis for the *computational context implementation* classes. These classes encapsulate the computation and algorithms related to the actual functionality of the protocol, i.e. producing the service provided by the protocol.

In order to make it possible to have several concurrent communication sessions each having its own computational context instance the *Master-Slave* pattern is applied [5]. A computational context implementation class acts as a *worker*. Worker class instances are created and terminated independently of each other. A worker is active only during the lifetime of a communication session. A *master* creates new worker instances whenever needed and keeps a record of active workers.

In the earlier specification phases the identification of a message receiver plays no important role. When a message is sent, its route to another protocol entity is determined from the message routes in a model. Now, routing functionality is required because there can be several active instances of the same class simultaneously. The functionality determines the proper receiver for an incoming message. Here, the *Message-Broker* pattern is applied. In this example, it is applied twice. In the first, routing functionality determines the type of the computational context, PC or IE, and in the second one, the worker instance within the type.

PDUs cannot just be sent from a protocol entity to the peer entity because there is no real direct physical peer-to-peer communication path between the two protocol entities. Therefore, the computational contexts in different network elements have to use another service, which provides a transparent data transmission service between the network elements. It's necessary to realize the virtual PDU communication on top of the existing lower layer service, and at the same time, it's desirable to preserve the notion of PDU communication in the worker classes. Application of the *Codec* pattern (aka *Peer-Proxy*) provides a solution [6]. A peer-proxy serves as a proxy for a peer entity. It intercepts the outgoing PDUs sent by a worker and encodes them producing byte strings. The encoded byte strings are then sent using the lower layer data transmission service. In case of an incoming PDU, the peer-proxy receives the incoming lower layer data service primitive and then decodes the contained PDU. Finally it sends the decoded PDU to the appropriate worker.

In this example, the computational context specific master, message-broker and peer-proxy functionalities have been combined into two classes, namely *PC_RED* and *IE_RED* (Routing-Encoding-Decoding). In addition, there are two subprotocols within the PCAP protocol (PC and IE) and implementation for the subprotocols should be kept as separate as possible. Computation context type specific routing (i.e. whether a message is a PC message or an IE message) is separated from PC RED and IE RED into the *MUX* class because routing is based only on information in messages. The need for the MUX class arises from the fact that there is only one service access point for the PCAP protocol. If there were two separate specific service access points for PC and IE subprotocols then the functionality of the MUX class would just be routing of incoming lower layer service primitives.

An example of one possible system architecture is illustrated in Figure 17: A system architecture.

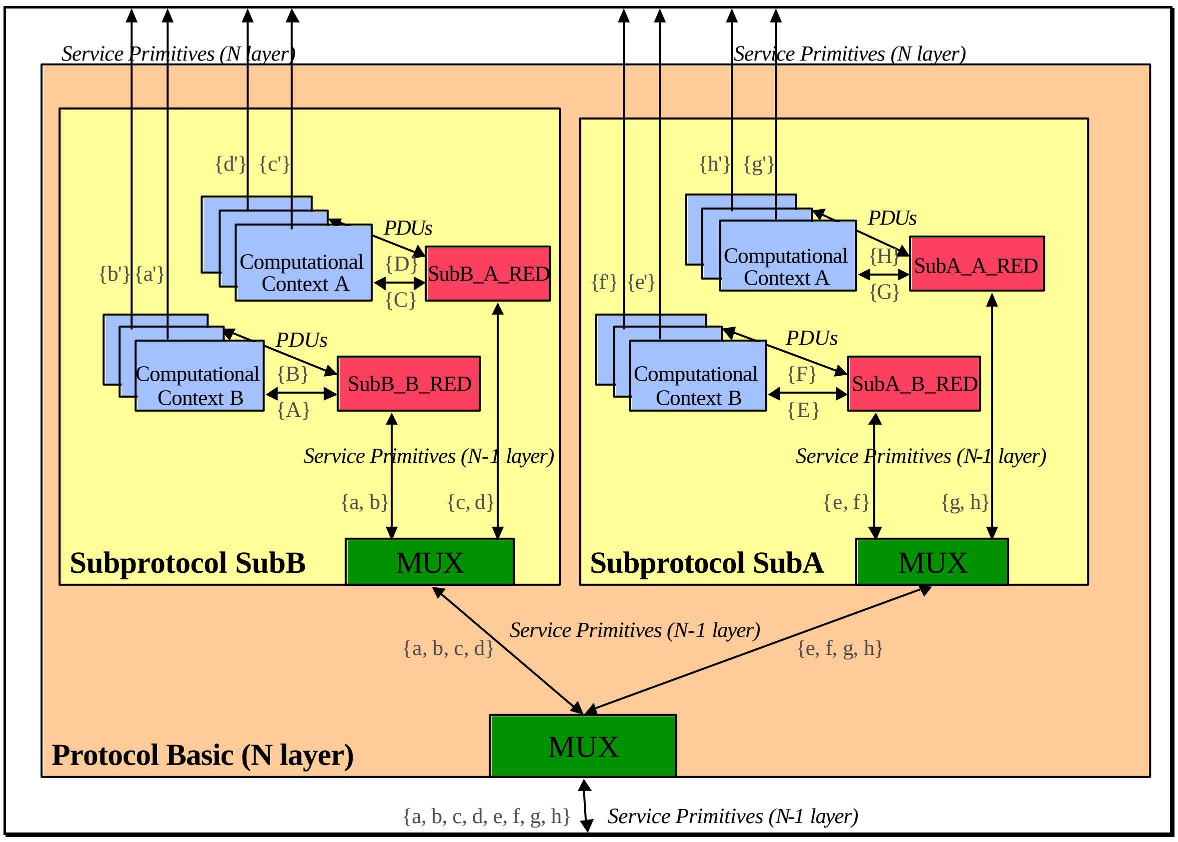


Figure 17: A system architecture

4.2 Interfaces in Implementation

To facilitate real communication between protocol entities located in different network elements the *Codec* pattern is applied (see 4.1 above). As a result, a peer-proxy needs to communicate with a lower protocol layer providing a data transmission service. This protocol layer has to use the provided service interface of a lower protocol layer.

For the computational contexts the PDU interfaces have been defined in the preceding phase. To enable the virtual PDU communication with the peer-proxy functionality, the peer PDU interface of each computational context is defined to the corresponding peer-proxy entity.

The *service primitives* exchanged with the underlying data transmission service carry the PDUs from a protocol entity in one network element to another protocol entity in another network element. The routing of messages is multilayered through using the RED and MUX entities. Therefore, usually the service primitive interface visible from the protocol entity to the environment is decomposed into several 'sub-interfaces' of RED and MUX entities inside the protocol entity. In Figure 17 the outermost service primitive interface is decomposed first into service primitive interfaces of the subprotocols and then further into service primitive interfaces specific for each computational context.

Figure 18 illustrates the PC_RED class and the interfaces defined for the class. All incoming messages are received via *mux_port*. The *pc_port* is used for PC service primitive exchange and *peer_proxy_port* is used for PDU exchange. Encoded outgoing PDUs are sent via the *sctp_port* in an SCTP service primitive.

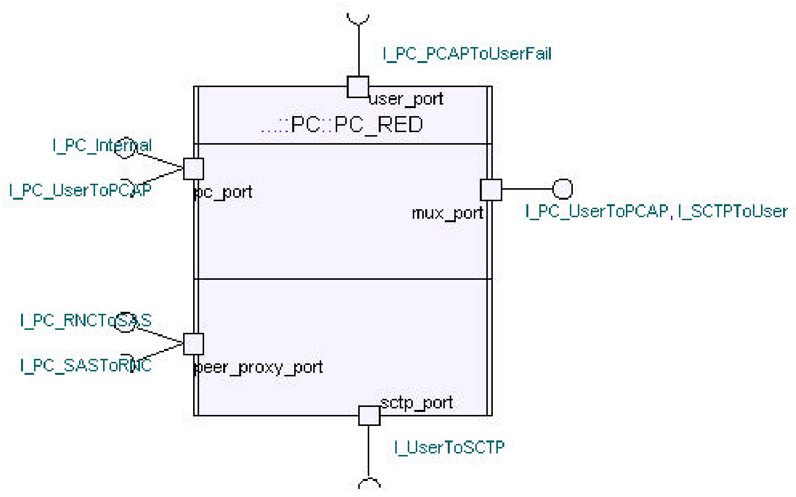


Figure 18: The PC_RED class

4.3 Architecture in Implementation

The architecture diagram describes the system configuration for automatic code generation. The classes defining the functional components are instantiated as parts. The interfaces between the classes are instantiated through the port instances defined for each part, and the communication between the ports is illustrated using connectors. Also the external interfaces are represented in the diagram.

Figure 19 illustrates the internal architecture of the PCAPServices. Only the PCService component and computational context are included in this Implementation model.

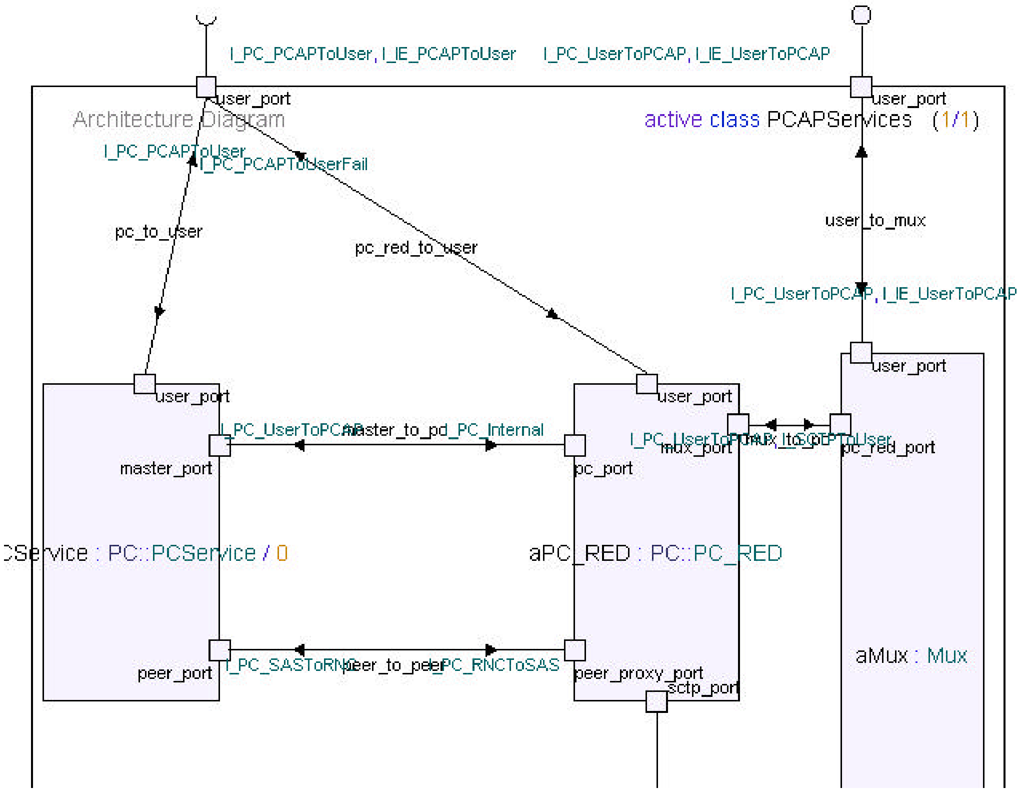


Figure 19: An implementation architecture

4.4 Behaviour in Implementation

The majority of the behaviour of the computational context has been defined already in the Service Distribution phase. In the Implementation phase the *Master-Worker* pattern is applied (see 4.1). As a result, the lifetime of a computational context instance is only one communication session. A worker is created, is active and finally determines to terminate itself. The behaviour of the RED and MUX entities shall be specified together with the details that were left vague in the Service Distribution phase.

Figure 20 shows how the behaviou r of a PCService class is augmented with dynamic aspects (object termination) and with error handling behaviour (timer). Figure 21 shows how PC_RED reacts when it receives the PC_req service primitive. The basic control flow looks the same as in the Service Distribution phase but details are added. The actual implementation details are hidden in the operators.

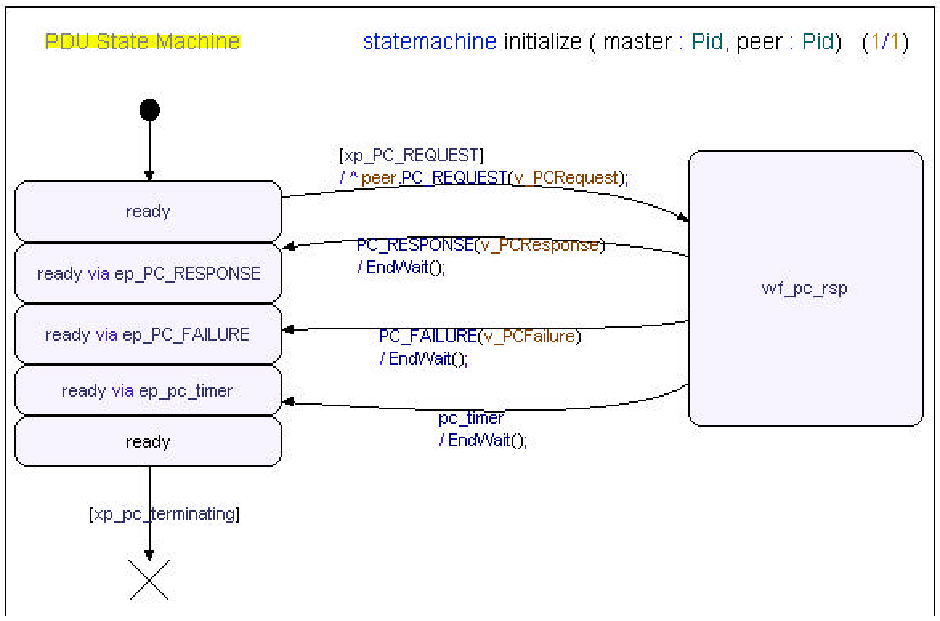
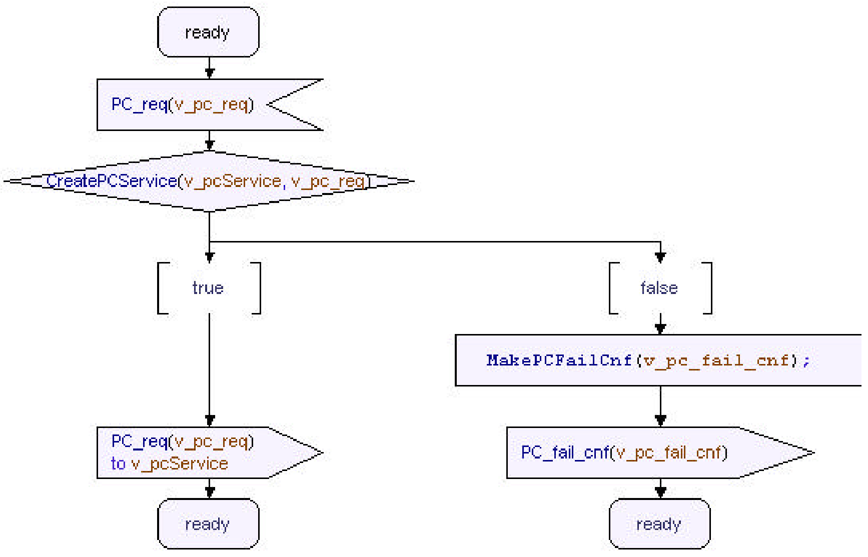


Figure 20: The state machine of the PCService class



ready
PC_req(v_pc_req)
CreatePCService(v_pcService, v_pc_req)
true
false
MakePCFailCnf(v_pc_fail_cnf);
PC_req(v_pc_req)
to v_pcService
PC_fail_cnf(v_pc_fail_cnf)
ready
ready

Figure 21: A part of the state machine of the PC_RED class

5. CONCLUSION

This paper has described a model-driven design method for protocol engineering, where a high -level requirement specification is refined step-by-step into a fully detailed implementation model. The high -level requirement model presents the functional requirements for the valid behaviour of the system as an executable finite state machine. The internal architecture of the functionality (or the behaviour), i.e. the software implementing the system, is described with service components. The interaction between the service components is encapsulated with well-defined interfaces. The behaviour of each service component is described with a finite state machine, and is refined step-by-step from a high-level, nondeterministic description into a detailed implementation model. Executable code for the target platform is generated automatically from the implementation model. In the refinement chain the tracebility between the models is validated with exhaustive simulation. The externally observable behaviour on each interface should remain the same throughout the refinement chain. In practice, the signal definitions of various interfaces are updated every once in a while, which has to be taken into account as a maintenance issue for the refined interfaces.

This method aims at improving the productivity of the design process and the quality of the software produced in it. Models on various abstraction levels enable evaluation of the basic ideas by simulation and provide fill-in skeletons for the subsequent models in the refinement chain. Analysing the core algorithms in very early design phases reduces costs and improves the software quality from the very beginning of the design process. Modularity enables reasonable and efficient resource allocation and supports maintenance and re-usability of the software components. Encapsulating the heuristics used in the design process and describing them as design patterns stabilizes various phases of the process and makes them repeatable.

Raising the abstraction level of the design work, i.e. moving from the implementation languages to specification and modelling languages, enables greater problem solving. The designer can concentrate on the essential - on the problem to be solved by her/his code.

APPENDIX I: BACKROUNDER ON UML 2.0

The Unified Modeling Language (UML) has enjoyed spectacular success since it was first standardized in 1997. It represents the end of the modeling wars and is being used across the board within the software industry. Initially, it was created to be a language for ‘specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system’.

During the five years since the standard was created, tool vendors and users have gained much experience with the language, and have learned to distinguish the most effective elements of the language from those parts which have not lived up to expectations. As tools implement new features that are not in the standard, users expect these features to be made part of the standard. A typical example of such a feature is executability of models, which enables early verification of system functionality without having to delve into application code.

Within the software business, five years is an eternity so it’s no surprise that new software trends have outgrown the capabilities of UML. In particular, the area of component-based development has caused problems for UML modelers; it has not been clear how to deal with component-frameworks such as COM+ and EJB, and also how to deal with the style of hierarchical decomposition of building blocks that typically occur in embedded systems development.

Two years ago, the OMG initiated the work on creating UML 2.0 – a new major revision of the most popular modeling language around – to take these and many other concerns into account. We are now starting to see the outcome of this work, and some of the new capabilities are described below (based on the UML 2.0 submission by the U2 Partners [10]).

In this research report, we have examined how to make use of these capabilities in practice as part of a case study on *protocol engineering*. The tool used in the case study is Telelogic Tau/Developer, which implements many of the described UML 2.0 features, including the capability to execute models.

A.1 Active and Passive Classes

Classes are easily the most widely recognized concept of UML. In the major revision, a great deal of effort has gone into making them more suitable for development of large-scale systems. This means that they are more adapted to the kind of component-based development that typically occurs within embedded systems development, and it also means they are more suitable to model component-based frameworks such as EJB and COM+.

For these kinds of systems, subsystems are often distributed in a network and execute concurrently with each other. In other words, each subsystem can be viewed as a system in its own right. In such systems, it is often beneficial to distinguish between *active* and *passive classes*, where active classes tend to describe the logical architecture of the system, while passive classes are used for describing data structures. Notationally, an active class uses the same notation as an ordinary class, but with vertical bars on the side (such as the active class VendingMachine as shown in Figure 22). Active classes run in threads of their own (i.e. are scheduled) while a passive class must be executed in the context of other classes, such as an active class. (The *main* function of a C or C++ program corresponds to the behavior of an active class representing the entire program, which in turn is normally scheduled by the operating system.)

It should be noted that active classes tend to have mostly receptions, i.e., the ability to receive asynchronous signals, rather than operations. Conversely, a passive class tends to have mostly synchronous operations. In particular for embedded systems, the use of active classes is predominant.

In order to deal with the complexity of developing large components concurrently in distributed teams, the use of clear interfaces between the different parts is imperative. In essence, each such component is treated as a building block that can then be put together with each other in specific ways.

A.2 Provided and Required Interfaces

Interface-based design has become increasingly important in modern software development, and many programming languages have incorporated the interface concept. Users want to be able to develop each part as a standalone entity that is independent of the other parts of the systems; for this to work it is necessary to express the contract between each part and its environment. In order to facilitate modeling of this, UML distinguishes between *provided* and *required interfaces.* A provided interface is represented by the traditional lollipop symbol and describes the services that a class implements, and a required interface describes the services that the class expects others to fulfill. Notationally, the required interface is very similar to the lollipop, except that the circle is replaced by a half-circle.

Note that a provided interface is shorthand for a class realizing an interface, while a required interface is shorthand for a class using an interface, i.e., it is also possible to model these interfaces using explicit relationships between the class and its interfaces. From the perspective of the class, it is not important to know which other building blocks are going to provide the functionality of its required interfaces; the services can be provided by anyone (not necessarily a class) that somehow realizes the required interfaces.

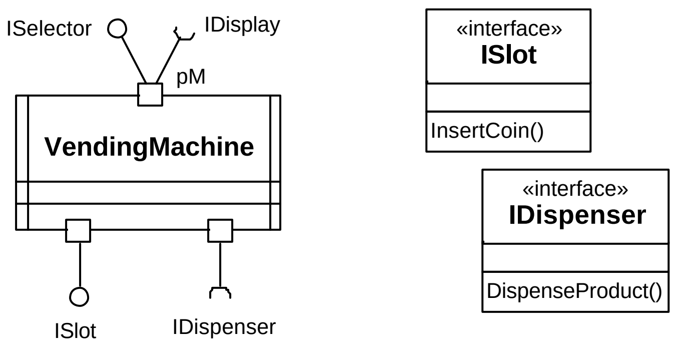


Figure 22: Provided and required interfaces

In Figure 22, a small example of an active class VendingMachine is shown. Here we see examples of both provided interfaces (ISelector and ISlot) and required interfaces (IDisplay and IDispenser). This class is viewed as a black box, and at this point, we cannot explain too much about its implementation; however, we know enough to be able to use the class, since the interfaces give the services that are required and provided. It is useful to couple this with information about how to use the interfaces, which can be done for example through sequence diagrams or protocol state machines. Either of these may be used to describe how messages are interchanged, and the order in which services may be invoked. The squares that are shown on the class are *ports*, which we will examine more closely in the next section.

Provided and required interfaces allow you to describe contracts between classes, in which case only classes with matching interfaces should be allowed to communicate with each other. The interfaces match if they are either of the same kind, or one interface is a subclass of the other.

A.3 Ports

The *port* has several different but related purposes. First, it can be used to group interfaces that have related functionality; in this regard it provides a view of the class that can be separately addressed. Secondly, a port acts as an interaction point, through which different classes can be connected together as parts of internal structures.

In most cases, you are only allowed to access the services of a class that has ports through its ports. The ports then act as holes in the shell of the class encapsulation through which it is possible to send or receive messages. Operations or attributes of such classes are not public; services of the class can only be accessed through the provided interfaces of the class. In Figure 22, the class has three ports; the top one is named pM, and has one required and one provided interface. In this case, communication over the port is bi-directional. In general, a port can be associated with any number of required and provided interfaces. The other two ports each have a single interface; one is a required interface (IDispenser) and the other is a providee interface (ISlot). These ports only support unidirectional communication, which means that calls can be sent to a class with a provided interface and return values received. For a class with a required interface, calls can be sent from the class to anyone providing the interface and return values received.

A.4 Internal Structure with Parts and Connectors

Hierarchical decomposition is a powerful and commonly used mechanism for structuring large and complex systems. To some extent, this is covered in the discussion about interfaces: modularity is an essential factor when a project is too big for only a few persons. Each class is viewed as a modular building block, which may be further broken down into smaller building blocks. In other words, a class may delegate its behavior to other classes that are parts of its internal structure. Each of those classes can be further broken down, and the recursion bottoms out when the class at the finest level of granularity only has behavior and no internal structure. It is possible to build arbitrarily large systems in clear structures using this approach.

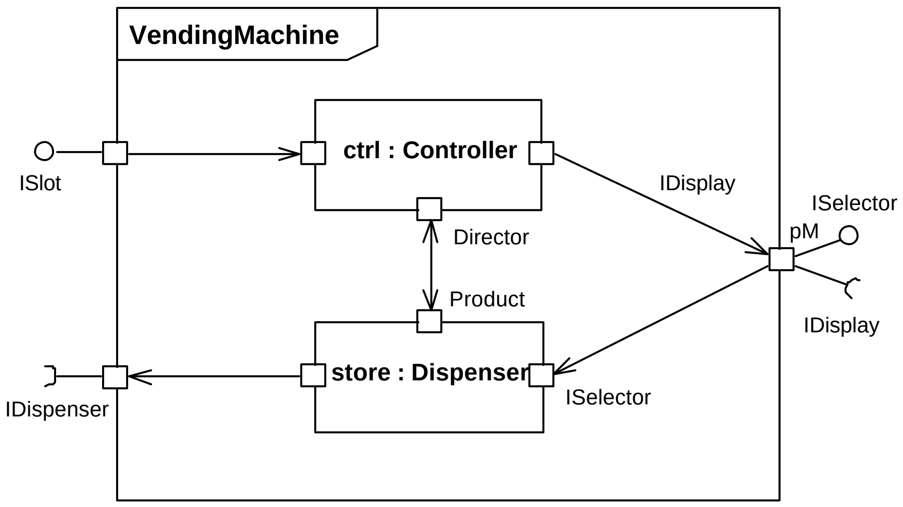


Figure 23: The internal structure of a class

The internal structure of a class is essentially made up of parts and connectors between the parts. Each *part* represents a usage of a class in the context of the container class, and the same class can be used as part of various contexts (i.e., internal structures), as is shown in Figure 23. This corresponds to the internal structure of the class that was shown in Figure 22. It also shows how the ports are used as connection points for the connectors. A *connector* is essentially a contextual association, which is only valid within the context where it is used, and the connectors of an internal structure describe the valid communication paths of the shown portion of the system. The classes may be connected in other ways as parts of other internal structures. The parts, which are named store and ctrl, respectively, depend on the class definitions Dispenser and Controller, which are not shown in this picture.

As the name implies, hierarchical decomposition relies on composition, and in Figure 24 the example from Figure 23 is shown using plain composition. These two views are complementary and express different information. The internal structure of a class only shows one layer of the composition hierarchy, and it is necessary to zoom into or out of the classes to look at other levels. Furthermore, the internal structure focuses on the communication between the different parts. In Figure 24, only the composite associations that are implied by Figure 23 are shown. This view is not very suitable to show the context specific information of an internal structure. The composition relationship between the classes implies that there are lifecycle dependencies between a container class and the parts of its internal structure; when a container class is created, then instances of the classes that are represented as parts are also created (depending on their multiplicities). Similarly, when an instance of a container class is deleted, then the instances that are represented as parts are also terminated.

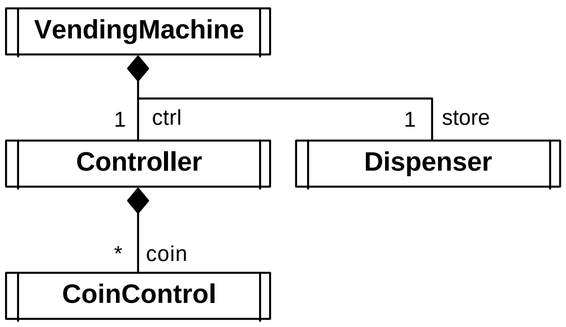


Figure 24: A composite structure

Each class is viewed as a building block and, just as it is possible to break it down into smaller pieces, it is equally possible to reuse it in a larger context to create even larger building blocks. This is also a common way to deal with legacy systems, which are then represented as building blocks with given interfaces that often cannot be changed.

A.5 Behavior Ports

In the black box view, as covered in a previous section, there is no way of knowing whether the communication received at a port is handled directly by instances of the class, or if they are delegated to other classes as part of its internal structure. In the white box view as represented by an internal structure, you can specify that a message sent to a *behavior port* should not be delegated to a part, but rather be handled by the container class itself. This distinction between ports is hidden in the black box view since it represents an implementation aspect of the class that should not be exposed. A behavior port is shown using a small state symbol that is attached to the square representing the port. A behavior port that is drawn entirely inside the class boundary represents a port that can only be accessed from within the class (i.e., from parts in its internal structure). Examples of behavior ports are shown in Figure 25.

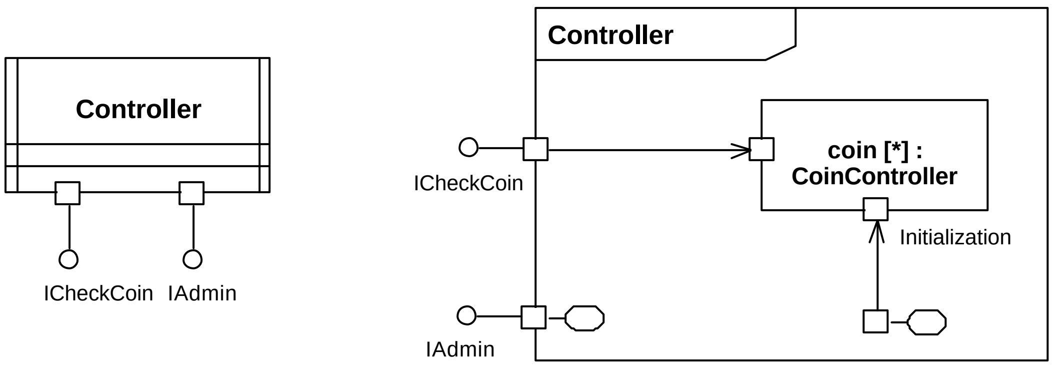


Figure 25: Behavior ports of a class

The definition of the class Controller is shown to the left, while its internal structure is shown to the right. The behavior port is tied directly to the behavior of the container class, which may be expressed for example through a state machine, an activity, or an interaction. In this case, the container class Controller is responsible for creating and initializing instances of the CoinController. It has a protected behavior port that is only used to pass initialization information to the created instances, and this initialization port of the CoinController is not accessible from outside Controller.

A.6 Behavior Descriptions

So far, we have only covered the new structural aspects of UML 2.0. However, the language has been significantly revised on the behavioral side as well, and in particular interactions and activities have received a significant overhaul. It's not necessary to study these new concepts in detail for this particular white paper.

Despite their popularity, interactions are quite underrated in UML. They can be used throughout the development lifecycle, and are equally good at expressing requirements, functionality, and tests. In addition, they are easy to understand, even for someone who does not know UML. Figure 5 and Figure 10 are examples of simple interactions. Unfortunately, the capabilities of interactions in UML 1.x were too limited for them to be really useful when dealing with larger systems. The most significant changes in UML 2.0 include:

- The ability to *reference* other interactions from within a sequence diagram, thereby avoiding the necessity to duplicate information in multiple interactions. Using this approach it is possible to quickly put together new behaviors based on already existing ones, for example when creating test suites.

- Expressing *variations* within sequence diagrams by enclosing one more messages with a frame; the variations include iterations, decisions, optionality, etc.; this reduces the number of sequence diagrams required to express functionality dramatically.

- *Decomposing* a lifeline into a new interaction that express message flows between the parts of the object representing the decomposed lifeline. This capability allows you to zoom into and out of interactions in much the same way that you can with decomposed classes.

State machines have not changed as much as other behaviors. The most significant change is probably the simplification of the metamodel (which is used to define the language), but this does not affect users much. However, it is now easier to define composite states, and these may also have entry and exit points. Such points work in much the same way as ports in that they disconnect the environment of the state from the internals of the state, and allow you to define specific points at which to enter or leave a state. As an example of a composite state, consider the Idle state in Figure 14, whose substates are shown in Figure 15.

A.7 Actions

Executable models is one of the catch-phrases of UML 2.0, and what makes it possible to execute models is the fact that actions have been specified to a level of granularity comparable with most programming languages. Note, however, that UML is not specified in such a way that it can immediately be used as a programming language; it is necessary to couple it with one or more *profiles* that close the different semantic variation points that give the language its flexibility, and it also helps to combine this with a data model having basic data types. Furthermore, such a profile should tie a concrete action syntax – notation – to the actions to make them accessible for most users, since the language itself provides no notation.

Actions include for example assignments, calls, loops and decisions, and correspond quite well with what is known as *statements* in other languages.

The actions that are defined were originally specified in the action semantics for the UML, which was a very late addition to UML 1.x, but have in UML 2.0 been integrated with activities to create a more homogeneous language. These actions can be used to describe the more detailed behavior of, for example, operations, state transitions or activities.

The main advantage of executable models is that they provide the capability to verify the correctness of a system before any code is produced and find errors much earlier in the development lifecycle. An executable model can be simulated – debugged – and it is also possible to apply various verification and validation techniques to the model. Equally important is the fact that most of the code, if not all, can be automatically generated from a model once you are satisfied that it works correctly. This makes it possible to focus on defining the functionality in the model, and letting code generators worry about the code necessary for memory allocation, distribution, etc.

Developing systems this way means that you will refocus your development efforts. Much more time will be spent during the analysis and design phases, where you will also be able to test your system repeatedly. At the same time, you reduce the time spent in implementation since most of the code is generated for you; the testing phase that normally follows the implementation phase is significantly reduced as much of the testing occurs earlier. Note that it is often quite expensive to find errors late in the development process, which means that there is a lot of time and money to be saved from being able to detect and fix bugs earlier.

A.8 Model Driven Development

Not all models have to be executable. In fact, most models that are created are not executable, and are there primarily to convey information between different stakeholders in a development process (i.e., to communicate intent). Most of the capabilities and flexibility of UML 1.x have been preserved in UML 2.0, but the language has been made more cohesive. At the same time, it has been extended to allow for better scalability when capturing the structure and behavior of larger systems, and for better precision when designing more detailed behavior.

UML 2.0 is designed to be suitable all through a development project, allowing you to model requirements, analysis, design, implementation, and testing. The term model-driven development implies that a model is at the centre of development, and is used as the basis for application development. The fact that the entire system is captured in a model makes it more accessible to the project members and less dependent on key architects, and also makes it easier for new project members to be phased into a project. Furthermore, it puts the focus on the functionality of the system rather than on the code that will need to be produced. To get the real benefits of model-driven development, however, both executing models and code generation play important roles, as they provide the means to build better systems faster.

REFERENCES

[1] 3GPP TS 25.305: "Stage 2 functional specification of UE positioning in UTRAN".

[2] 3GPP TS 25.453 V5.2.0: "UTRAN Iupc interface PCAP signalling (Release 5)".

[3] 3GPP TR 21.905: "Vocabulary for 3GPP Specifications".

[4] ITU-T. Recommendation I.130, “Method for the Characterization of Telecommunication Services Supported by an ISDN and Network Capabilities of an ISDN, extract from the Blue Book.

[5] F. Buschmann, et al., "Pattern-oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1996

[6] J. Ellsberger, D. Hogrefe, A. Sarma, "SDL, Formal Object-oriented Language for Communicating Systems", Prentice Hall Europe, 1997

[7] J. Pärssinen et al., "UML for Protocol Engineering - Extensions and Experiences", Proceedings for 33rd International Conference on Technology of Object-Oriented Languages. 2000

[8] M. Luukkainen, A. Ahtiainen, ”Compositional Verification of SDL descriptions”, In SAM’98, 1st Workshop of the SDL Forum Society on SDL and MSC, 1998.

[9] S. Leppänen, M. Luukkainen, ”Compositional Verification of a Third Generation Mobile Communication Protocol”, In Proc. International Workshop on Distributed System Validation and Verification, IEEE, 2000.

[10] U2 Partners, “Unified Modeling Language: Superstructure, version 2.0”, OMG ad/2003-01-02, www.u2-partners.org, 2003.