TUCS

Dubravka Ilić | Sari Leppänen |
Elena Troubitsyna | Linas Laibinis

Towards Automated Model-Driven Development of Distributed Communicating Systems and Communication Protocols

Turku Centre *for* Computer Science

TUCS Technical Report
No 829, July 2007

Towards Automated Model-Driven Development of Distributed Communicating Systems and Communication Protocols

Dubravka Ilić

TUCS, Åbo Akademi University, Department of Information Technologies, Joukahaisenkatu 3-5 A, 5th floor
20520 Turku, FINLAND

Sari Leppänen

Nokia Research Center, Computing Architectures Laboratory,
P.O. Box 407, 00045 Helsinki, FINLAND

Elena Troubitsyna

Åbo Akademi University, Department of Information Technologies
Technologies, Joukahaisenkatu 3-5 A, 5th floor
20520 Turku, FINLAND

Linas Laibinis

Åbo Akademi University, Department of Information Technologies
Technologies, Joukahaisenkatu 3-5 A, 5th floor
20520 Turku, FINLAND

TUCS Technical Report
No 829, July 2007

Abstract

Model-driven development has gained increasing acceptance in the engineering community. Via abstraction and gradual model transformation, it offers an efficient way to cope with complexity of modern software-intensive systems, typical examples of which are distributed telecommunicating systems and communication protocols. However, variety of models representing the system structure and behaviour from different viewpoints and at different levels of abstraction raise the question of model consistency and their adherence to the predefined architectural rules. In this paper we formalize a development flow of distributed telecommunicating systems and communication protocols as an architectural profile in UML. We specify and formally verify this profile. The profile allows us to check adherence of models to the predefined architectural rules. Furthermore, by formalizing and verifying intra- and inter-consistency rules, we ensure that the models do not contradict to each other. We use the B Method as our formal framework. The presented work establishes a basis for automating model-driven development of telecommunicating systems and communication protocols.

Keywords: B Method, consistency of UML models, formal methods, refinement, UML profiles

TUCS Laboratory
Distributed Systems Design

1. Introduction

MDD (Model Driven Development) [1] has emerged as the paradigm aiming at ensuring cost-effective and time-efficient software development. It is gaining increasing acceptance in the software engineering community. MDD is design-centric, i.e., it focuses primarily on modelling the system functionality and behaviour rather than the technology to implement it. At the time when the technology is changing rapidly, MDD allows the developers to reuse their previously developed solutions and, as a result, reduce costs and time of developing new applications. Moreover, it enables a fast integration of emerging technologies into the existing systems.

The ideas of MDD are implemented via UML (Unified Modelling Language) [2]. It is a graphical modelling notation used to create system models. Modelling with UML typically starts from abstract, high-level models, which are then iteratively transformed into more detailed models. However, validating a large variety of produced models with respect to the given architectural rules is a recognized problem when modelling with UML. In UML2 [3, 4], the architectural rules can be defined in a systematic way using the built-in light-weight extension mechanism called *profiles*. The profiling mechanism allows us to specify a new modelling language by defining the architectural rules for the system under development. These rules represent the modelling concepts and constraints on them in a particular domain. With the support of a proper tool, we can use the defined architectural rules for ‘driving’ the development process and automatically checking whether the produced models conform to them. Therefore, UML profiles can provide a solid basis for increasing the level of automation in software development.

In this paper we introduce the *Lyra profile* – a UML2 profile that defines the architectural rules for the Lyra design method [5, 6]. Lyra is a model-driven and component-based design method for development of distributed communicating systems and communication protocols. It has been developed at Nokia Research Center and applied in large-scale industrial development projects. Lyra consists of four consecutive development stages. At each development stage, a system can be described from different viewpoints. These viewpoints are visually represented by different types of UML2 models. Hence, the Lyra development flow results in a large set of models, which raises the question of validating the models against the predefined architectural rules and managing model consistency.

Ensuring consistency of Lyra models is a two-fold task. On the one hand, we need to ensure intra-consistency of the models, i.e., consistency among artefacts specifying different aspects of the system on the same development stage. On the other hand, we should guarantee inter-consistency of models, i.e., consistency among modelling artefacts from the different development stages. In this paper we propose an approach to formal verification of model consistency in Lyra. We use the description of the Lyra design method given in the form of the Lyra profile to derive general patterns for UML2 models created at different stages of Lyra development and express intra- and inter-consistency rules for them. Then, we define Lyra models as formal specifications in the B Method [7] – a formal framework for modelling complex software-intensive systems. Each Lyra model together with the corresponding intra- and inter-consistency rules is represented by a B model. Hence, in our approach the B Method serves as a common semantics for UML2 models. In this respect, our approach to ensuring consistency of UML2 models is similar to the approach based on defining a common semantics of UML presented by Derrick et al. in [8].

Since both MDD and B adopt the top-down development paradigm, it is natural to describe the model-driven UML2-based Lyra development in B. In our approach, the B development starts from an abstract system specification, which simulates creating Lyra models in the order defined by the design method. The abstract specification contains the models from the first Lyra development stage and intra-consistency rules defined for them. It is transformed into more detailed specifications by correctness preserving steps called refinements. The refinement process allows us to structure complex intra- and inter-constancy requirements and handle them in a stepwise manner, by specifying and verifying only part of them at a time. Then, we add the remaining requirements subsequently. The resulting refined specifications represent more detailed models and their intra-consistency rules. In addition, they specify the inter-consistency rules defined between models at two subsequent stages.

The obtained B specifications and refinements are formally verified by the use of an automatic tool support provided for B – AtelierB. The formal verification ensures intra- and inter-consistency of the corresponding UML2 models, thus establishing the basis for automatic verification of the Lyra design flow.

The paper is structured as follows. In Section 2 we briefly introduce UML profiling principles. Section 3 describes the Lyra design method via an example. Section 4 describes the design method in the form of the Lyra profile, which is introduced through its basic concepts and creating principles. In Section 5 we define the notion of consistency in Lyra. Section 6 continues by giving a short introduction to our modelling framework – the B Method. In Section 7 we describe our approach to ensuring intra- and inter-consistency in Lyra by formal specification and refinement in B. Section 8 discusses the related work. Finally, in Section 9 we conclude with the overview of the proposed approach and the future work.

2. UML profiles

The latest version of UML - UML2 - significantly differs from its previous versions. The most considerable structural change is the division of the UML2 specification into two complementary specifications: Infrastructure and Superstructure. They define respectively the foundation language constructs and the user-level constructs required for UML2. UML2 Infrastructure [3] is assumed to be extensively reused when creating various metamodels. For instance, Meta-Object Facility (MOF) reuses it to provide the ability to model metamodels and UML2 Superstructure [4] reuses it to define UML metamodel itself.

In UML2, *profiles* are the built-in light-weight extension mechanism which allows customization of UML for different domains. Profiles can be used to extend a MOF-based metamodel, e.g., the UML metamodel, for a specific context, domain or purpose. Profiles are only allowed to contain tagged values, stereotypes, constraints and data types [4]. Stereotypes represent variations of existing modelling elements (e.g., UML2 metaclasses) with the same form (having the same attributes and relationships) but with a modified intent [4]. A stereotype can have additional constraints on the base metaclass it extends as well as tagged values containing additional information for a stereotyped element. Tagged values are defined as properties of the form name-value, where the name is used as a tag. In UML2, these properties can be attached to the introduced stereotypes by marking them as attributes inside a class representing a new stereotype.

The profiling mechanism is defined by the package *Profiles* in UML2 Infrastructure. As it is not a first-class extension mechanism of UML, it does not allow modifications of existing metamodels [4]. This implies that the newly introduced stereotypes, meta-attributes, and associated constraints cannot contradict with the reference metamodel; it is impossible to take away any of the metamodel constraints, but it is possible to add new constraints that are specific to a profile. In short, the reference metamodel is considered always as a "read only" specification. This implies that the specialized semantics should not contradict with the semantics of the reference metamodel. This restriction on using the UML profile mechanism guarantees, e.g., that any CASE-tool compliant with the UML2 metamodel can be used for constructing models conformant with a UML2 metamodel based profile.

As a part of a UML2 profile, it is not allowed to have an association between two stereotypes or between a stereotype and a metaclass, unless it is a subset of the existing association in the reference metamodel [4]. In other words, according to the above-mentioned profiling principles, the introduced association should be related to an association of the same type in the reference metamodel. Moreover, the multiplicity ranges of the introduced association should match the corresponding multiplicities of the association in the reference metamodel. Described associations provide a convenient and intuitive way to model the introduced restrictions and constraints on a profile. Further, such associations could also be expressed using OCL (Object Constraint Language) [9] in UML profiles. In fact, UML2 Infrastructure proposes two methods to achieve the effect of new (meta)associations: (1) adding new constraints within a profile that specialize the usage of some associations of the reference metamodel, or (2) extending the Dependency metaclass by a stereotype and defining specific constraints on this stereotype.

Various UML profiles have been recently introduced for different purposes. For instance, OMG proposes UML profiles for CORBA [10], for Schedulability, Performance and Time [11], for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms [12] etc. The *Lyra profile*, which this paper introduces, is built based on the Lyra design method described in the following section.

3. Overview of Lyra design method by an example

Lyra [5, 6] is a service-oriented and model-based design method for the development of distributed communicating systems. It has been developed in Nokia Research Center by integrating the best practices and design patterns established in the domain. The method has been successfully applied in several large-scale industrial development projects.

Lyra has four main stages: *Service Specification, Service Decomposition, Service Distribution and Service Implementation*. The *Service Specification (SS)* stage defines the services provided by the system and the different types of users of these services. A service is a functionality that the system provides. In this stage we define the externally observable behaviour of the system services on the corresponding user interfaces. In the *Service Decomposition (SDe)* stage the abstract model produced in the previous stage is decomposed into a set of service components and logical interfaces between them. This stage yields the logical architecture of the service implementation. In the *Service Distribution (SDi)* stage the logical architecture of services is distributed over a given platform architecture. This results in a physical architecture of a distributed communicating system. Finally, in the *Service Implementation* stage the structural elements are integrated into the target environment. In this stage we arrive at a model which can be used as, e.g., a source for automatic code generation. A detailed description of the Lyra method can be found elsewhere [5, 6].

We exemplify the Lyra design method by modelling a positioning system of Third Generation Partnership Project (3GPP) [13]. The system provides the positioning service for calculating the physical location of a given user equipment in a mobile network. A detailed informal description of the service can be found in [13].

As a modelling language for describing the positioning system we use UML2 [3, 4], although the Lyra design method is generic with respect to modelling languages and tools.

Models at the Service Specification stage. Our first development stage – Service Specification – starts from creating Domain Model. It is a UML2 use case model that specifies the service PositionCalculation within Positioning system and the type of its user – User, as represented in Fig. 1.

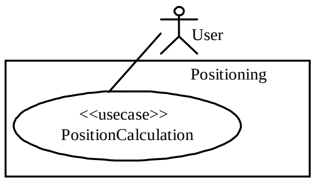


Fig. 1. Domain Model of the positioning system at the SS stage

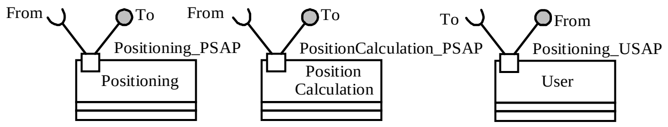


Fig. 2. Communication Context of the positioning system at the SS stage

At this stage we also create Communication Context model, where Positioning system and PositionCalculation service are defined as active classes, as shown in Fig. 2. To model interfaces via which a system service is provided, we attach UML2 ports to the active classes. In Lyra these ports are called *Provided Service Access Points* (PSAPs). They are defined for the classes Positioning and PositionCalculation as Positioning_PSAP and PositionCalculation_PSAP respectively. In the Communication Context model we also specify the external class for the system user – User – with the attached port. However, since this port models an interface of a service user, it is called *Used Service Access Point* (USAP) (see, e.g., Positioning_USAP in Fig. 2).

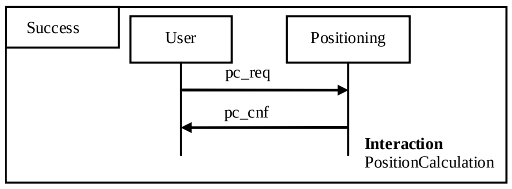
The UML2 interfaces on all specified PSAPs define the signals and signal parameters of the system-user communication. For instance, the interfaces To and From of PositionCalculation_PSAP class are specified as follows:

```
interface To_ PositionCalculation_PSAP {
   public signal pc_req (part PCReqParam);
}
```

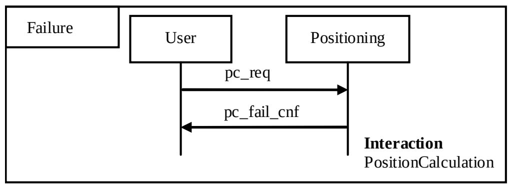
```
interface From_ PositionCalculation_PSAP {
   public signal pc_cnf (part PCCnfParam);
   public signal pc_fail_cnf (part PCFailCnfParam)
}
```

Here PCReqParam, PCCnfParam, and PCFailCnfParam are the abstract data structures encapsulating actual signal parameters.

The descriptions of the interfaces and the valid order of the signals are visually represented by interactions in Signalling Scenario models. PositionCalculation interaction (shown in Fig. 3) comprises two Signalling Scenario models. They describe the signals (pc_req, pc_cnf, pc_fail_cnf) between the communicating entities in case of service Success (Fig. 3a) and Failure (Fig. 3b).



a)



b)

Fig. 3. The Signalling Scenario models of the positioning system at the SS stage

The communication between the PositionCalculation service and its user is described in the PSAP Communication state machine, as shown in Fig. 4. The positioning request pc_req received from the user should always be confirmed – by the signal pc_cnf in case of success and by pc_fail_cnf otherwise.

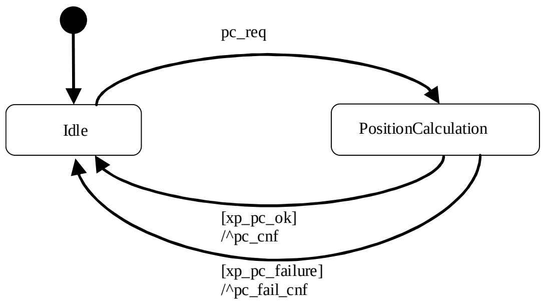


Fig. 4. PSAP Communication of the positioning system at the SS stage

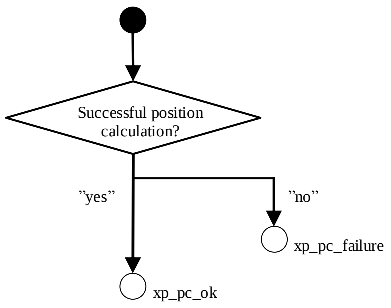


Fig. 5. PositionCalculation Substate Machine at the SS stage

In the PSAP Communication model, the PositionCalculation state is composite. At this level of abstraction we define the behaviour for its substates by a non-deterministic Substate Machine. Such a model non-deterministically determines the success or failure of the service execution, as shown in Fig. 5.

Models at the Service Decomposition stage. To provide the position calculation service, the positioning system uses services provided by some external service providers. For instance, to provide the positioning service, at first Radio Network Database (DB) should be requested to send the information on an approximate location of the user equipment (UE). This information is then used to contact UE. Then, another external service provider – Reference Local Measurement Unit (RefLMU) – is requested to provide the reference measurements to calculate the exact location of UE. This information is handled by the positioning Algorithm server to produce the final estimation on the UE location.

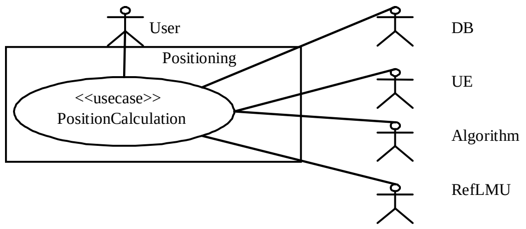


Fig. 6. Domain Model of the positioning system at the SDe stage

At the SDe stage these external service providers are introduced into the previously developed system models. In Domain Model we introduce the corresponding actors for DB, UE, RefLMU and Algorithm. They are associated with the PositionCalculation use case, as represented in Fig. 6. Correspondingly, the Communication Context model now contains the external classes representing DB, UE, RefLMU and Algorithm. Each of these classes should have its own PSAP describing the communication with the system service (e.g., DB_PSAP in Fig. 7). Moreover, for the active classes Positioning and PositionCalculation we define USAPs via which the external services are used. Each active class should have USAP for each external class. For instance, we define DB_USAP for both Positioning and PositionCalculation classes as shown in Fig. 7.

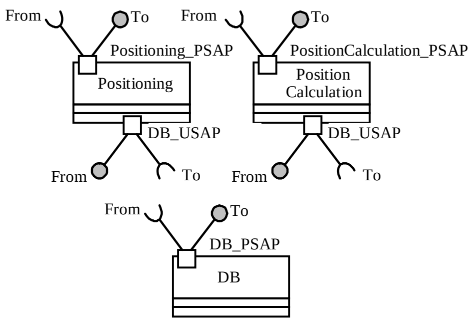


Fig. 7. Excerpt from the Communication Context model of the positioning system at the SDe stage

The PositionCalculation service comprises several subservices. These subservices are modelled as the subuse cases: LMU_Measurement, DB_Enquiry, UE_Enquiry, and Algorithm_Invocation in the Decomposition Diagram of the positioning system, as represented in Fig. 8.

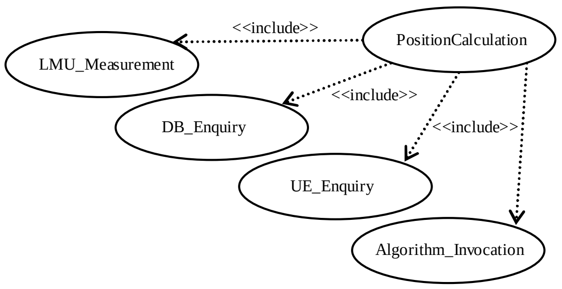


Fig. 8. Decomposition Diagram for the PositionCalculation service at the SDe stage

The order of the subservice execution is defined in the Signalling Scenario models using the interaction references (ref). Each interaction reference represents a set of Signalling Scenario models for some subservice. The subservice execution order is determined by the order in which these references appear in the Signalling Scenario model for the PositionCalculation service, as shown in Fig. 9.

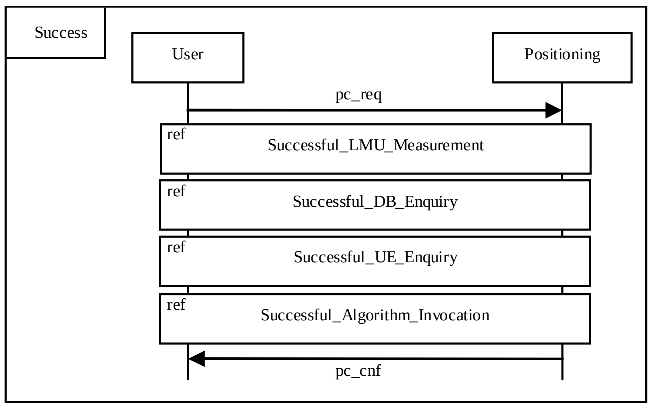

Success
User
Positioning
pc_req
ref Successful_LMU_Measurement
ref Successful_DB_Enquiry
ref Successful_UE_Enquiry
ref Successful_Algorithm_Invocation
pc_cnf

Fig. 9. The Signalling Scenario model for the successful execution of the PositionCalculation service at the SDe stage

We represent the logical architecture of the positioning system by Architecture Diagram, as shown in Fig. 10. It describes the logical structure of the active class Positioning (defined in the Communication Context model in Fig. 7). In Architecture Diagram, the active class Positioning is called a *system component*. A system component is composed of several logical elements. Each of them encapsulates a part of the service functionality and is called a *service component*. The service components of the Positioning system component are shown in Fig. 10.

The part of the service functionality which handles the communication with DB, while requesting an approximate location of the user equipment, is encapsulated within the service component DBHandler. Similarly, the service component UEHandler manages the communication with the corresponding user equipment. RefLMUHandler handles the communication with the external service provider RefLMU and computes the intermediate measurement results. At last, the service component AlgoHandler conducts the final calculations at the requested user equipment position.

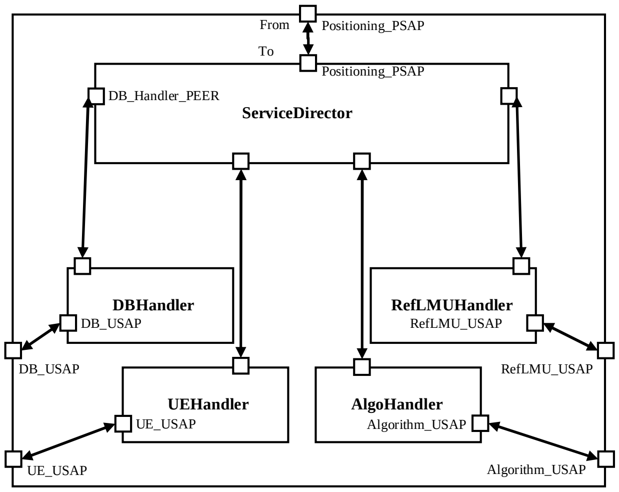


Fig. 10. Instantiation of the Positioning logical architecture

The execution flow of the introduced service components is managed by an architectural element called ServiceDirector. It processes service requests and orchestrates the execution of the service components. The behaviour of ServiceDirector is described in a hierarchical way. The top-most state machine of ServiceDirector is the PSAP Communication model in Fig. 4.

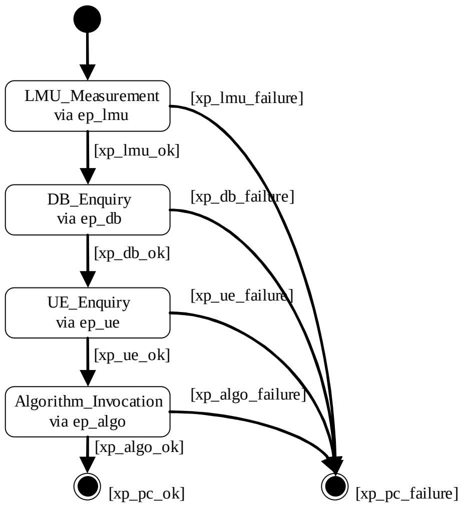


Fig. 11. Execution Control of the positioning system at the SDe stage

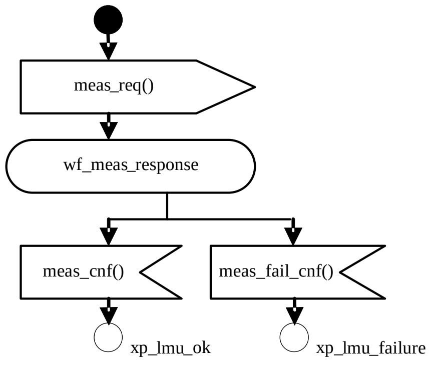


Fig. 12. LMU_Measurement USAP communication at the SDe stage

In the SDe stage the composed state PositionCalculation from the PSAP Communication model is decomposed into a set of substates in the Execution Control state machine, shown in Fig. 11. These substates are: LMU_Measurement, DB_Enquiry, UE_Enquiry, and Algorithm_Invocation.

The substates of the Execution Control state machine are further refined. They describe either some internal computation in the substates or USAP communication that triggers the execution of a particular service component. The refined behaviour is represented in the corresponding substate machines. For instance, USAP communication in the substate LMU_Measurement is described in Fig. 12.

Models at the Service Distribution stage. The SDi stage focuses on distributing system components over a given network architecture. The positioning system should be distributed over two network elements: Positioning_RNC (Radio Network Controller) and Positioning_SAS (Stand-alone Assisted Global Positioning System Serving Mobile Location Center). The distributed positioning service is represented by the domain models for each network element. The Domain Model for Positioning_RNC and Domain Model for Positioning_SAS are shown in Fig. 13a and 13b respectively.

When modelling the service distribution over the network element Positioning_RNC, Positioning_SAS becomes an external service provider. Therefore, it is modelled as an actor together with the existing external service providers DB and UE. Similarly, when modelling the service distribution over Positioning_SAS, we represent Positioning_RNC as an actor together with RefLMU and Algorithm.

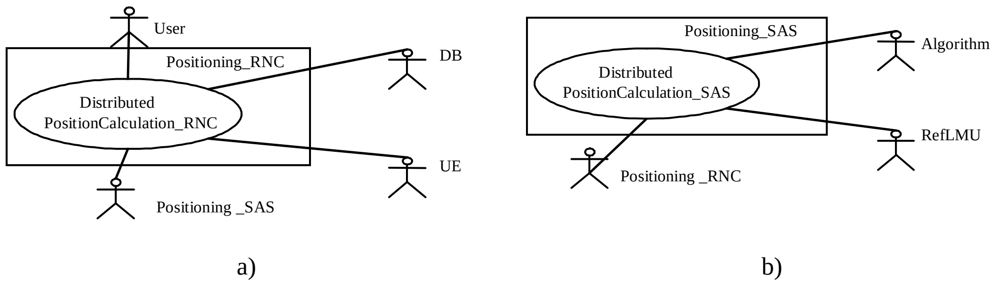


Fig. 13. Domain Model of the positioning system at the SDi stage

The Communication Context model of the positioning system (shown in Fig. 14) reflects the service distribution represented in the domain models (Fig. 13). It defines the following active classes: DistributedPositionCalculation_RNC, Positioning_RNC, DistributedPositionCalculation_SAS, and Positioning_SAS.

The communication between the network elements is defined via the ports Positioning_SAS_PEER and Positioning_RNC_PEER of the classes Positioning_RNC and Positioning_SAS (see Fig. 14) respectively. Observe that both DistributedPositionCalculation_SAS and DistributedPositionCalculation_RNC classes have the same ports as the corresponding classes Positioning_SAS and Positioning_RNC.

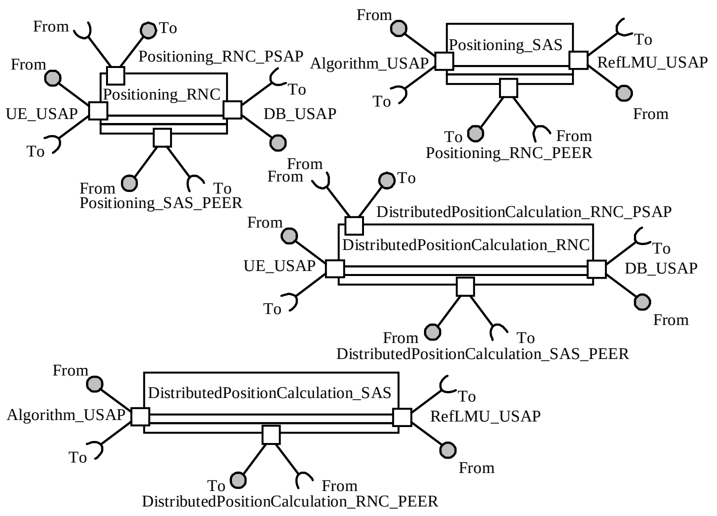

Positioning_RNC
Positioning_RNC_PSAP
UE_USAP
DB_USAP
Positioning_SAS_PEER
Positioning_SAS
Algorithm_USAP
RefLMU_USAP
Positioning_RNC_PEER
DistributedPositionCalculation_RNC
DistributedPositionCalculation_RNC_PSAP
DistributedPositionCalculation_SAS_PEER
DistributedPositionCalculation_SAS
DistributedPositionCalculation_RNC_PEER
From
To

Fig. 14. Excerpt from Communication Context of the positioning system at the SDi stage

Since the positioning system services and subservices are distributed over different network elements, their decomposition is represented by two distinct Decomposition Diagrams – for Positioning_SAS and for Positioning_RNC.

By introducing the network elements Positioning_RNC and Positioning_SAS, we map the logical architecture of the positioning system to the physical network architecture. It is represented by Architecture Diagrams for both network elements. Architecture Diagram for the Positioning_SAS element is given in Fig. 15.

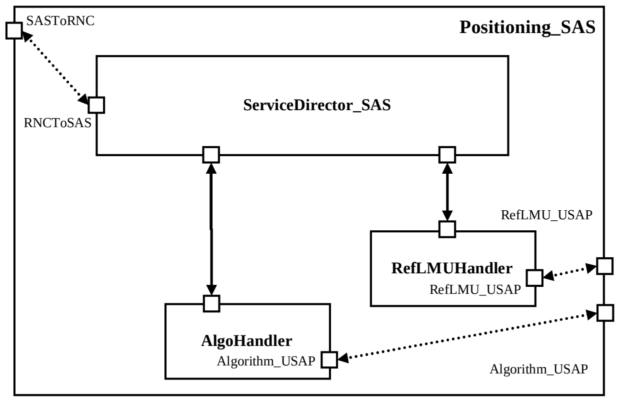


Fig. 15. Logical architecture of the Positioning_SAS network element

While mapping the logical architecture of the positioning system to the actual network architecture, we distribute the service components and ServiceDirector across the network elements Positioning_SAS and Positioning_RNC. The distributed ServiceDirector of the network element Positioning_SAS is called ServiceDirector_SAS (in Fig. 15). It controls the service components AlgoHandler and RefLMUHandler of the network element Positioning_SAS, as shown in the state machine Execution Control in Fig 16. ServiceDirector_SAS also manages the communication with Positioning_RNC.

Similarly, ServiceDirector_RNC of Positioning_RNC controls the service components DBHandler and UEHandler and the communication with Positioning_SAS. In addition, ServiceDirector_RNC handles the communication with the system user.

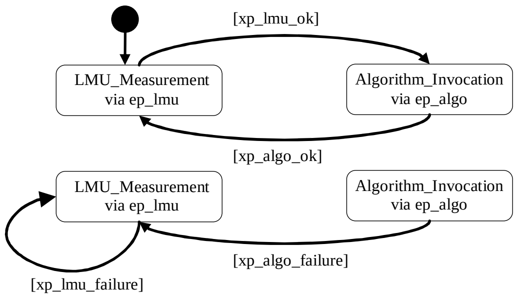

[xp_lmu_ok]
LMU_Measurement via ep_lmu
Algorithm_Invocation via ep_algo
[xp_algo_ok]
LMU_Measurement via ep_lmu
Algorithm_Invocation via ep_algo
[xp_algo_failure]
[xp_lmu_failure]

Fig. 16. Execution Control state machine of ServiceDirector_SAS

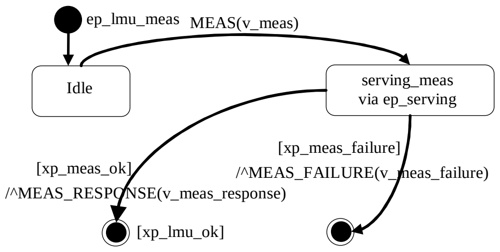

ep_lmu_meas MEAS(v_meas)
Idle
serving_meas via ep_serving
[xp_meas_failure]
/^MEAS_FAILURE(v_meas_failure)
[xp_meas_ok]
/^MEAS_RESPONSE(v_meas_response)
[xp_lmu_ok]

Fig. 17. PEER Communication of ServiceDirector_SAS

We specify the communication between Positioning_RNC and Positioning_SAS via corresponding PEERs by PEER Communication state machines. For the network element Positioning_RNC, the PEER Communication state machine coincides with the state machine describing USAP communication with RefLMU in Fig. 12. The PEER Communication state machine of the network element Positioning_SAS is represented as the PSAP Communication state machine in Fig. 17.

The fourth Lyra stage – Service Implementation – focuses on implementing low level details (e.g., data encoding and decoding, routing of messages etc.) on the top of the already existing architecture. Here we omit its detailed description which can be found elsewhere (e.g., [6]).

While presenting the development of the positioning system, we have already described some elements of the Lyra profile. In the next section we present the entire profile in detail.

4. The Lyra profile

4.1. Lyra profiling principles

The development of the Lyra profile was inspired by the earlier work by Selonen and Xu [14, 15] on defining a profile for capturing architectural rules and constraints relevant to the specific product line platforms. Selonen and Xu introduced a concept of an *architectural profile* [14, 15], which relies on the UML_1.5 profile mechanism. Architectural profiles are extended UML profiles specialized for describing architectural constraints and rules for a given domain [15, 16]. Selonen and Xu use extended UML profiles to enable adding constraints on inherited meta-associations between user-defined stereotypes. (Let us note, that this is not allowed in UML_1.5 profiles). These extended profiles contain two parts: the standard UML metamodel part showing the subset of the metamodel that is being extended, and the extension part showing the stereotypes and the inherited meta-associations and other constraints. The extension part describes the valid relationships between the architectural concepts: classifiers, interfaces, and dependencies between them. The actual architecture checked against the profile must satisfy the constraints implied by the profile, i.e., it should not have other structures except the ones explicitly described in the profile. The dependencies used in the extension part of an architectural profile represent visualizations of the constraints that could also be expressed, e.g., using OCL [14, 15]. Such visualizations are easy for a software architect to read and understand. While introducing the Lyra profile, we use the corresponding visualizations in a similar way as it is done in the architectural profiles.

4.2. Concepts of the Lyra profile

The Lyra profile represents the basic reference model that allows the designers to check correctness and completeness of the Lyra design models. For instance, we can verify correctness of all models of the positioning system at each Lyra stage with respect to the profile.

To introduce the Lyra profile, we use UML2 as our description language. This allows us to avoid unnecessary redefinitions since most elements of Lyra models reuse existing UML2 notions.

The aim of the Lyra profile is to tailor the existing UML2 metamodel to the Lyra design method. We customize the UML2 metamodel by introducing specific stereotypes. They allow us to use the Lyra concepts in modelling and add corresponding semantics to the metamodel. Namely, each Lyra stereotype allows us to use a specific Lyra element while modelling either system structure or behaviour.

While presenting the profile, we show Lyra stereotypes as extensions of the corresponding UML2 meta-classes. For clarity, we show only the associations between stereotypes and omit the corresponding meta-associations between the extended meta-classes as described in [15].

4.2.1. Introducing stereotypes for modeling the system structure

To model the system structure, we use Domain Model, Decomposition Diagram, Communication Context, and Architecture Diagram, as described in the previous section. The elements of these models are instances of either the existing meta-classes from UML2 metamodel or the Lyra stereotypes. We focus only on those models that rely on some Lyra stereotype.

To define Domain Model, we need a concept of a specialized actor representing either a user of the modelled service or its provider. Hence, we introduce a stereotype called ***LyraActor***. *LyraActor* is an external entity that interacts with the system. This is an abstract concept that can be instantiated either as ***ServiceUser*** or ***ServiceProvider***. *ServiceUser* is an external user of the system, whereas *ServiceProvider* is an external provider of some other service the system might use. In our positioning system example the instance of *ServiceUser* is User and the instances of *ServiceProvider* are DB, UE, RefLMU and Algorithm (shown in Fig. 6).

Architecture Diagram in Lyra is a UML2 component diagram whose basic elements are components. Lyra introduces different types of these components designated by new stereotypes – ***SystemComponent*** and ***ServiceComponent***.

SystemComponent is a structural model element, which encapsulates a logical, independent piece of a system specification. Hence, it can be developed in isolation and later integrated into a larger system. *SystemComponent* consists of *ServiceComponents*. They are logical model elements, which encapsulate the service behaviour and have specific functionalities. In the positioning system example, an instance of *SystemComponent* is Positioning system component shown in Fig. 10. It consists of four instances of *ServiceComponent*: DBHandler, UEHandler, RefLMUHandler, and AlgoHandler. Each of these instances encapsulates a part of the positioning service functionality.

SystemComponent interacts with its environment by requesting or providing a service. The Lyra profile introduces the abstract stereotype ***AccessPoint*** to specify different communication points between *SystemComponent* and its environment. *AccessPoint* can be instantiated either as ***SAP*** (Service Access Point) or ***PeerAP*** (Peer Access Point).

SAP is a communication point through which a system may either provide its services to the external users or use the services provided by external service providers. *SAP* through which a system provides its service is called ***PSAP*** (Provided Service Access Point), whereas *SAP* through which a system uses other services is called ***USAP*** (Used Service Access Point). In our example, the positioning system provides its position calculation service to the system user through PositionCalculation_PSAP, as shown in Fig. 7. However, to provide this service, the positioning system also uses (through DB_USAP) a service provided by the network database component.

PeerAP is a communication point between a set of distributed service components. For instance, the network elements Positioning_SAS and Positioning_RNC, which represent the actual distribution of the positioning service, communicate through Positioning_SAS_PEER and Positioning_RNC_PEER, as shown in Fig. 14.

4.2.2. Introducing stereotypes for modeling the system behaviour

As described in the previous section, we model system behaviour by using Signalling Scenarios and several different types of state machines: Execution Control, PSAP Communication, USAP Communication, PEER Communication, and Internal Computation. Elements of these models are instances of the corresponding meta-classes from the UML2 metamodel. However, we introduce Lyra stereotypes to represent the types of the state machines defining the hierarchical structure of behaviour, shown in Fig. 18.

The top-most state machine is an instance of the stereotype ***PSAPCommunication***. It describes the communication through PSAP of the owning *ServiceComponent*, i.e., the communication with the service user. For instance, the PSAP Communication state machine of the position calculation service shown in Fig. 4 describes the communication with the service user via PositionCalculation_PSAP from Fig. 2.

According to a received user request, a PSAP Communication state machine calls (i.e., invokes) an Execution Control state machine, which is an instance of the stereotype ***ExecutionControl***. It defines the execution flow of the subservices required to produce a response to a user request. For instance, the Execution Control state machine of the positioning system (represented in Fig. 11) defines the execution order of the needed subservices: LMU_Measurement, DB_Enquiry, UE_Enquiry, and Algorithm_Invocation.

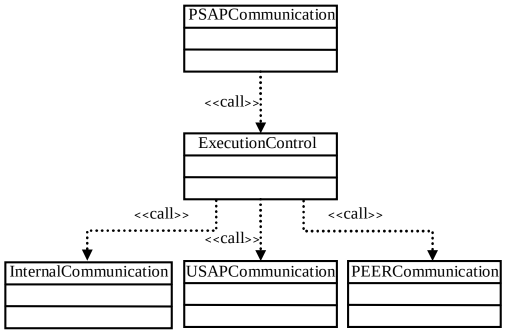


Fig. 18. Hierarchical structure of behaviour

The Execution Control state machine can invoke different types of behaviour depending on the way in which the subservices are implemented. Namely, if the implementation of a subservice involves some internal computations, which are not directly related to the service interfaces, the Execution Control state machine calls an Internal Computation state machine that is an instance of the stereotype ***InternalComputation***. However, if some subservices are provided by the external service providers then the Execution Control state machine calls a USAP Communication state machine. This is an instance of the stereotype ***USAPCommunication*** that describes the USAP communication of the owning *ServiceComponent*, i.e., the communication with an external service provider. For instance, the USAP Communication state machine represented in Fig. 12, describes the communication with an external service provider – RefLMU. Finally, if some part of the service implementation has been distributed to a remote physical location then the Execution Control state machine calls a PEER Communication state machine. It is an instance of the stereotype ***PEERCommunication***. A PEER Communication state machine describes the communication through PEERs of the owning *ServiceComponent*. For instance, the PEER Communication state machine, represented in Fig. 17, specifies the communication through Positioning_SAS_PEER of the Positioning_SAS network element with another network element Positioning_RNC.

In general, the PSAP, USAP and PEER Communication state machines together with the Execution Control and the Internal Computation state machines specify the overall service logics. They all are instances of the stereotype ***ServiceBehaviour***. *ServiceBehaviour* constitutes one part of the overall *ServiceComponent* behaviour. Another part – ***ServiceComponentBehaviour*** – is not considered to be a part of the service logics. It encapsulates all internal implementation-specific functionalities, like dynamic process management and routing of incoming messages. This is a part of the communication protocol, presentation of which we omitted in our positioning system example.

The summary of the Lyra profile is given in Fig. 19.

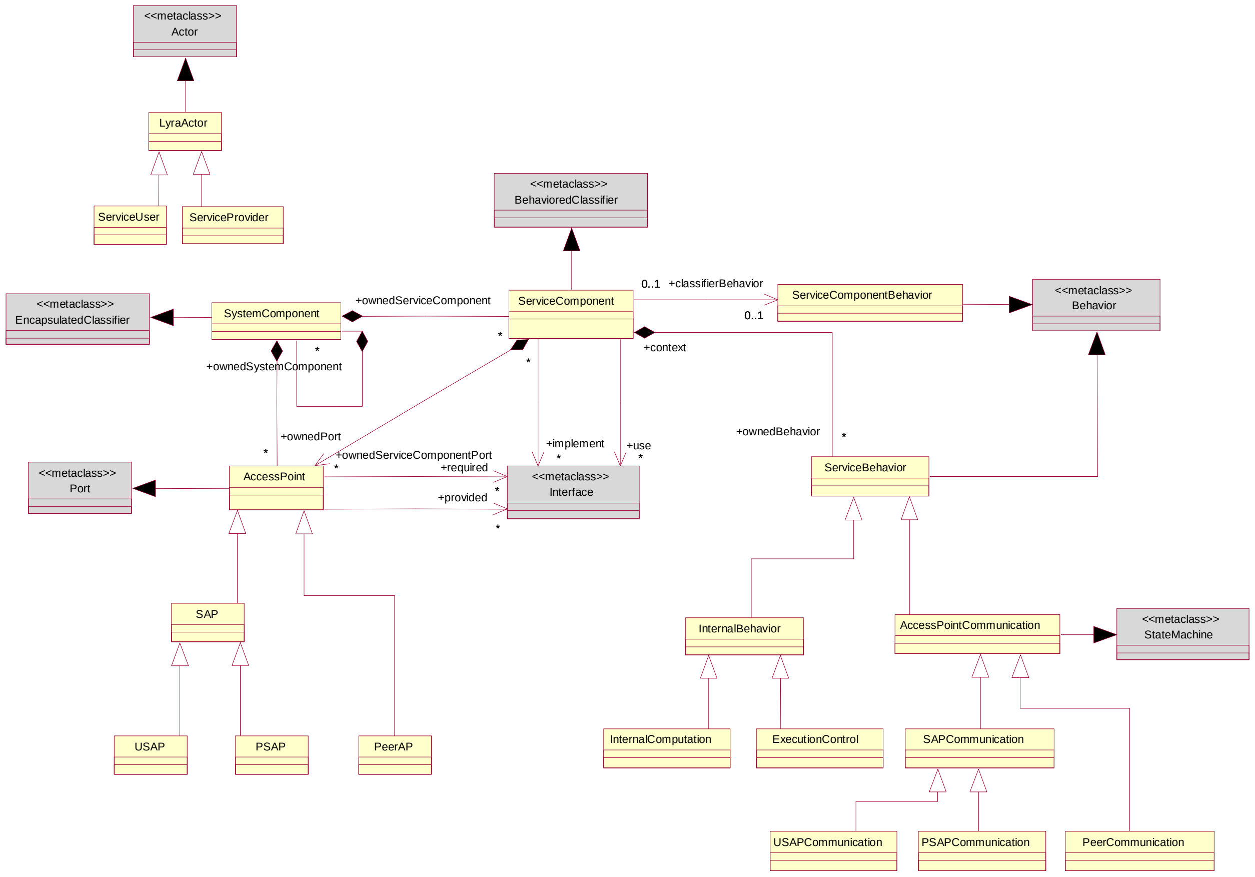


Fig. 19. Summary of the Lyra profile

5. Defining consistency in Lyra

The Lyra design method adopts the top-down development paradigm. Development starts from a high level of abstraction. The models at each subsequent stage represent the system at lower levels of abstraction, i.e., they specify the required functionality in more detail. This raises a problem of ensuring model *consistency*, throughout the system development. In other words, we have to guarantee that each properly defined model is not contradictory with already created models. We call a model *properly defined* if it satisfies the model presentation rules, i.e., the structural requirements imposed on the modelling elements.

Ensuring consistency is a two-fold task. On the one hand, a model should be consistent with the models at the same development stage. On the other hand, it should be also consistent with the models from the previous development stages. The consistency between the concepts specifying different aspects of the system structure and behaviour on the same development stage is known as *intra-consistency* [17]; whereas the *inter-consistency* [17] is defined as the consistency among modelling concepts from different development stages.

The Lyra profile presented in the previous section allows us only to ensure that created Lyra models are properly defined, i.e., that their structure conforms to the one defined in the profile. Defining consistency in the Lyra profile is, however, a difficult task. Although one could express intra-consistency rules as OCL constraints on the profile elements, it would still require referencing those UML2 meta-classes extended by the profile stereotypes. This would complicate the process of creating OCL constraints. Furthermore, Lyra is based on stage-specific development. Expressing inter-consistency rules for different Lyra stages would require either:

- annotating the existing profile elements to designate different stages and then add OCL inter-consistency constraints on the top of the already existing intra- consistency constraints, or
- creating a metamodel for each Lyra stage and again using OCL to express inter-constancy.

In both cases defining consistency in the Lyra profile would be complex and tedious.

Next we propose a formal approach to achieving intra- and inter-consistency in Lyra. We start from deriving general forms of consistency rules between Lyra models.

Models at the Service Specification (SS) stage. The system development starts from creating Domain Model describing the system services and their users. Its general form is given in Fig. 20a. To be properly defined, Domain Model should satisfy certain structural constraints. For instance, an association in the Domain Model can be created only if the corresponding actor and use case have been created first.

From Domain Model we derive the formal system structure represented in the Communication Context diagram. The general form of this model is shown in Fig. 20b. To be consistent with the previously created Domain Model, Communication Context should satisfy a number of intra-consistency rules.

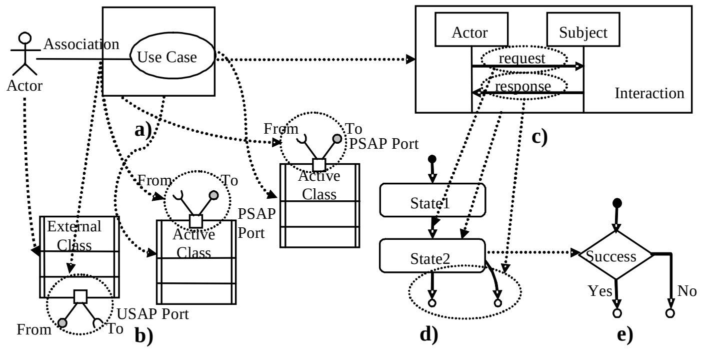


Fig. 20. The design flow of the SS stage

Table 1 shows an excerpt from the list of intra-consistency rules for the models in SS stage. Specifically, it shows part of the intra-consistency rules for the Communication Context model.

Table 1. Excerpt from the list of intra-consistency rules for Communication Context at SS stage

Rule	
1.1 1.2	One active class is created for the system which is defined in Domain Model. The name of the system is the name of the active class.
2.1 2.2	For each use case in Domain Model an active class is defined. The name of the class is the same as the name of the corresponding use case.
3.1 3.2	For each actor in Domain Model one external class is created. The name of the actor in Domain Model becomes the name of the external class.
4.1 4.2	The association between the actor and the system defines PSAP on the corresponding active class. The name of the port on the active class corresponding to the system is obtained according to the rule: <name of the system>_PSAP (i.e., <name of the active_class>_PSAP)
5.1 5.2 6.1 6.2	The association between the actor and the use case defines PSAP on the corresponding active class. The name of the port on the active class corresponding to the use case is obtained according to the rule: <name of the use case>_PSAP (i.e., <name of the active_class>_PSAP) The association between the actor and the system defines USAP on the corresponding external class. The name of the port on the external class is obtained according to the rule: <name of the system >_USAP

The next model at the SS stage – Signalling Scenario (Fig. 20c) – gives an informal description of the communication between a system service and its user(s). The communication is defined in terms of interactions. Each interaction is a set of the Signalling Scenario models defined for a particular system service.

Formally, the communication between a system service and its users is expressed in the PSAP Communication model (Fig. 20d), which is a UML2 state machine. In general, a PSAP Communication model has two states: the idle state and the composite state. The composite state is obtained from the interactions defined in the Signalling Scenario models. Transitions between the idle and the composite states specify the messages exchanged within the same Signalling Scenario models.

The main computation states in the PSAP Communication model are composite. The behaviour of the service on the level of substates is defined in the corresponding Substate Machine models (Fig. 20e). At the SS stage, Substate Machine also non-deterministically models success or failure of service execution.

Models at the Service Decomposition (SDe) stage. To provide a system services, the system usually relies on the services of some external service providers. Their explicit representation is introduced into the system model at the SDe stage. The external service providers are represented as new actors associated with the system services in Domain Model (Fig. 21a). To ensure that Domain Model at the SDe stage is consistent with Domain Model at the SS stage, we should guarantee that, after introducing external service providers, the elements of the model introduced at the SS stage remain unchanged. In other words, we should ensure inter-consistency between the models of those two stages.

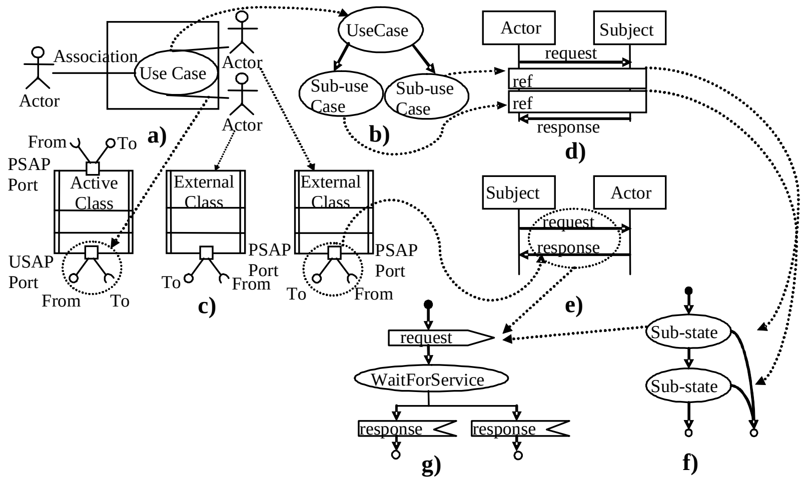


Fig. 21. The design flow of the SDe stage

Table 2. Excerpt from the list of inter-consistency rules for Communication Context at SDe stage

Rule	
1	Each active class created at SS stage should remain the active class in the Communication Context at SDe stage.
2	Each external class created at SS stage should remain the external class in the Communication Context at SDe stage.
3	PSAPs on active classes created at SS stage remain unchanged.
4.1 4.2	USAP is added to the active class corresponding to the system for each newly added actor in Domain Model at SDe stage. The name of the USAP is obtained according to the rule: <name of the added actor>_USAP
5.1 5.2	USAP is added to the active class corresponding to the use case for each newly added actor in Domain Model at SDe stage. The name of the USAP is obtained according to the rule: <name of the added actor>_USAP

At the SDe stage we rely on the intra-consistency rules defined for the SS stage. For instance, while creating Communication Context (Fig. 21c), we again define external UML2 classes for the actors introduced in Domain Model at the SDe stage. Each external class obtains its own PSAP, describing communication with the system service. Furthermore, each association between the system service and an external service provider is modelled as a USAP attached to the associated active classes. Let us observe that the elements introduced in Communication Context at the SS stage should remain unaffected, i.e., we should ensure inter-consistency between these models on SS and

SDe stages. An excerpt from the list of the inter-consistency rules for the Communication Context model at the SDe stage is shown in Table 2.

The decomposition of the system service into subservices is depicted in Decomposition Diagram (Fig. 21b). This is an additional model appearing at the SDe stage. Decomposition Diagram is actually a use case model showing the subuse cases that should be executed to provide the system service.

By defining the subservice execution order we complete the behavioural specification of a decomposed service. We augment the Signalling Scenario models created at the SS stage by adding interaction references (denoted as ref in Fig. 21d) representing a set of Signalling Scenario models (Fig. 21e) for each subuse case. These scenarios describe the communication between the system subservices and the external service providers. The subservice execution order is then defined by the order in which the references appear in the augmented Signalling Scenario (Fig. 21d).

At the SDe stage, the PSAP Communication model is refined to explicitly model the behaviour on the level of subservices. The composite state, modelling the actual service execution in the PSAP Communication model, is decomposed into a set of substates in the Execution Control state machine (Fig. 21f). The substates of the Execution Control state machine correspond to the subservices. The internal computation in the substates and the communication between the subservices are modelled for each substate in the corresponding Substate Machine (Fig. 21g).

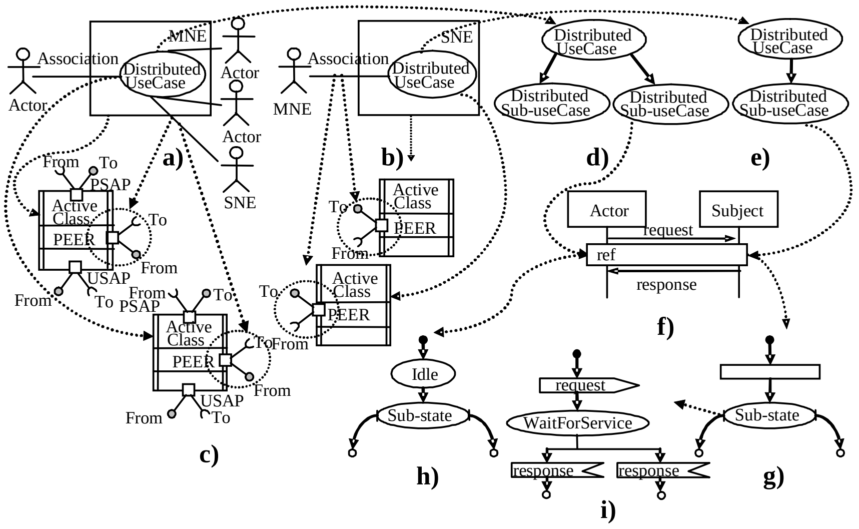


Fig. 22. The design flow of the SDi stage

Models at the Service Distribution (SDi) stage. The SDi stage focuses on distributing decomposed system services over a given platform architecture. The elements of Domain Model from the previous stage remain unchanged. However, they are now associated to the underlying platform and referred to as network elements. The network element that communicates with the user is called the Main Network Element (MNE), while the other network elements are called Secondary Network Elements (SNE). Since the system service distributed on each network element uses only services of the external providers allocated to that particular element, Domain Model at SDi stage should be defined for each of the network elements from their own viewpoints. This means that, when defining Domain Model for the MNE (Fig. 22a), we model the rest of the network elements as actors. Similarly, when defining Domain Model for SNE (Fig. 22b), we model the MNE and the other existing SNEs as actors.

The resulting set of the inter-consistency rules for Domain Model at SDe stage is shown in Table 3. The similar rules are defined for each model.

Table 3. Excerpt from the list of inter-consistency rules for Domain Model at SDi stage

Rule	
1.1	The system created in Domain Model at SDe stage is split into separate network elements in the SDi stage.
1.2	For each network element, new Domain Model is created.
1.3	The name of the system is obtained according to the rule: <Name of the system_Name of the network element>
2.1	Each use case created at SDe stage is distributed in the Domain Models at SDi stage across different network elements.
2.2	The name of the use case is obtained according to the rule: <Distributed_Name of the use case_Name of the network element>
3	Actors from the Domain Model at SDe stage are associated with different network elements and become a part of different Domain Models in SDi stage.
4.1	In each Domain Model for a network element, all the other network elements become actors associated with the system.
4.2	The name of the actor is the same as the name of the network element it is representing.

Communication Context (Fig. 22c) defines the active classes for all distributed services and corresponding network elements upon which they are distributed. The external classes defined at the previous Lyra stage remain unchanged. The associations from Domain Model define the interfaces on USAPs and PSAPs of the classes corresponding to the network elements. The communication between distributed services is defined via the PEER interfaces attached to the corresponding network elements.

Distribution of the decomposed functionality of the system is defined by the Decomposition Diagram models. Since the system services and subservices may be distributed on different network elements, Decomposition Diagram should represent the system decomposition from the individual viewpoints of each network element. This means that we should create Decomposition Diagram for the MNE (Fig. 22d) and Decomposition Diagram for the SNE (Fig. 22e).

The Signalling Scenario models (Fig. 22f) for the distributed services introduce interaction references for the distributed subuse cases. They describe the PEER communication between the parts of the distributed service.

The Execution Control state machine defined in the previous Lyra stage remains the same. However, Substate Machine attached to its composite distributed state is replaced with the new Execution Control machine (Fig. 22g) defining the distributed functionality in a remote location. It is defined from the viewpoint of the MNE. Additionally, the new PSAP Communication state machine (Fig. 22h) needs to be defined for the distributed service from the viewpoint of the SNE.

The composite states in the Execution Control machine are further specified by the corresponding Substate Machine models (Fig. 22i).

To summarize, the overall Lyra design flow is guided by the requirements imposed on its modelling elements: 1) each model is created according to certain structural requirements; 2) models within one stage are created according to the defined intra-consistency rules; 3) models at each subsequent development stage preserve the inter-consistency rules.

We show how to ensure consistency in Lyra by formalizing models and the intra- and inter-consistency rules defined above. The next section gives a brief introduction into our modelling framework – the B Method.

6. The B Method

The B Method [7, 18] (further referred to as B) is an approach for the industrial development of highly dependable software that has been successfully used in the development of several complex real-life applications [19]. The tool support available for B provides us with the assistance for the entire development process with a high degree of automation in verifying correctness. For instance, Atelier B [20], one of the tools supporting the B Method, has facilities for automatic verification and code generation. The high degree of automation in verifying correctness improves scalability of B and speeds up the development.

In B, a specification is represented by a module or a set of modules, called Abstract Machines. The common pseudo-programming notation – Abstract Machine Notation (AMN) – is used to construct and formally verify them. An abstract machine encapsulates a state and operations of the specification and has the following general form:

MACHINE	*Name*
SETS	*Set*
VARIABLES	*v*
INVARIANT	*I*
INITIALISATION	*Init*
OPERATIONS	*Op*

Each machine is uniquely identified by its *Name*. The state variables of the machine are declared in the **VARIABLES** clause and initialized in the **INITIALISATION** clause. The variables in B are strongly typed by constraining predicates of the **INVARIANT** clause. The constraining predicates are conjoint by conjunction (denoted as $\wedge$). All types in B are represented by non-empty sets and hence set membership (denoted as $\in$) expresses typing constraint for a variable, e.g., $x \in TYPE$. Local types can be introduced by enumerating the elements of the type, e.g., *TYPE = {element1, element2,…}* in the **SETS** clause. The operations of the machine are atomic, meaning that once started, they cannot be interrupted until finished. They are defined in the **OPERATIONS** clause. The operations are specified as the guarded operations of the form:

Operation = **SELECT** cond **THEN** body **END**

Here *cond* is a state predicate, and *body* is a B statement describing how the state variables are affected by the operation. If *cond* is satisfied, the behaviour of the guarded operation corresponds to the execution of its *body*. If *cond* is false at the current state then the operation is disabled, i.e., cannot be executed.

B statements that we are using to describe the computation in operations have the following syntax:

$$\begin{aligned} S \;==\; & x := e \mid x, y := e1, e2 \mid S1 \,;\, S2 \mid S1 \parallel S2 \mid \\ & x :\in T \mid \text{ANY } z \text{ WHERE cond THEN } S \text{ END} \mid \ldots \end{aligned}$$

The first three constructs – assignments and sequential composition – have the standard meaning. The remaining constructs allow us to model parallel and nondeterministic behaviour in a specification. The detailed description of the B statements can be found elsewhere (e.g., [18]).

B also provides structuring mechanisms for modularization. It allows handling the complexity of development by describing parts of the specification in separate machines. Here we use **EXTENDS** mechanism to incorporate these separate machines into the overall specification. When machine M1 extends machine M2, written as **EXTENDS** M2 in the definition of M1, it means that M2 is included as part of the machine M1. Its state is part of the state of M1. Moreover, all of the operations of M2 become operations of M1.

The semantics of B is based on the weakest precondition calculus [21]. If *S* is a B statement and P a predicate representing the postcondition, i.e., a set of states which can be reached after performing the B statement, then [*S*]P represents the weakest precondition that guarantee P after executing *S*. The weakest precondition rules for a subset of B statements are defined as follows:

$$\begin{aligned} &[\mathit{skip}]\, P \Leftrightarrow P \\ &[x:=E]\, P \Leftrightarrow P(x/E) \\ &[\mathit{S1} \parallel \mathit{S2}]\, P \Leftrightarrow [\mathit{S1}]\, P \text{ and } [\mathit{S2}]\, P \\ &[\textbf{ANY } z \textbf{ WHERE } \mathit{cond} \textbf{ THEN } S \textbf{ END}]\, P \Leftrightarrow \forall\, z\, (P \Rightarrow [S]\, P) \end{aligned}$$

They are used for verifying correctness of B specifications.

To ensure correctness of a B machine, we should verify that the initialization preserve the invariant and that the invariant is valid, which means that there are some possible machine states satisfying it. In other words, initialisation statement *Init* must always guarantee the machine invariant *I*:

$$[\mathrm{Init}]\ I \Leftrightarrow \mathrm{true}$$

Moreover, to establish correctness of a B specification, we should verify that every operation Op_i also preserves the invariant *I* when invoked under some precondition $cond_i$:

$$I \wedge \mathrm{cond}_i \Rightarrow [\mathrm{body}_i]\ I$$

Here body_i is the body of the operation Op_i.

The formal development in B is based on *stepwise refinement* [22]. While developing a system by refinement, we start from an abstract formal specification and transform it gradually into an implementable program by a number of correctness preserving steps, called refinements. The result of a refinement step in B is a machine called **REFINEMENT**. Its structure coincides with the structure of the abstract machine. In addition, it explicitly states which machine it refines.

In this paper we extensively use *data refinement* – a general form of refinement, which allows us to change the state space of a machine. To replace abstract data structures with the refined ones, we define the refinement relation (linking invariant) that explicitly states the connection between the newly introduced variables and the variables that they replace. The refinement relation constitutes a part of the invariant of the refining machine.

To ensure correctness of a refinement, we should verify that initialization and each operation of the refining machine refine the initialization and the corresponding operations of refined machine. Since the refinement relation is a part of the invariant of the refining machine, it suffices to ensure that the initialization and each operation of the refining machine satisfy this invariant.

While developing a system by refinement, it is often needed to introduce new variables while leaving the existing data structure unchanged. This is a specific form of data refinement called *superposition refinement* [22]. It also allows introducing new events which describe computations on these new variables.

The B tool support provides assistance in verification of B models. The verification can be completely automatic or user-assisted. In the former case, the tool generates the required proof obligations and discharges them without user's help. In the latter case, the user proves certain proof obligations using the interactive prover provided by the tool.

In the next section we demonstrate how to use specification and refinement in B to verify the consistency of Lyra models.

7. Formal verification of consistency

In Section 5 we derived informal consistency requirements. The informal requirements form the basis for formalizing Lyra models and consistency rules in B.

Ensuring intra-consistency of Lyra models in B. To ensure intra-consistency between the models in Lyra we should verify that models at one development stage:

- satisfy *model presentation rules*, i.e., the constraints expressing how to properly define model elements, and
- are *not contradictory* with each other.

To verify these properties, we first represent each Lyra model as a B machine of a general form given in Fig. 23.

The name of the machine corresponds to the name of a Lyra model and is followed by the acronymic name of the stage, i.e., SS, SDe or SDi. The variables of this machine correspond to model elements and their presentation rules are expressed as its invariant. Each operation simulates creating an element of the model. Namely, for each element, the corresponding **Create_ModelElement** operation represents creating the element according to the model presentation and the intra-consistency rules.

```
MACHINE         Model_Stage
EXTENDS         < Previously created model >
VARIABLES       < Names of model elements >, Model_Stage_Status
INVARIANT       < Model presentation rules >
INITIALISATION
    < Initialise the variables for model elements > || Model_Stage_Status:=Empty
OPERATIONS
Start_Model_Stage =
    BEGIN
          Model_Stage_Status:=Creating
    END;
Stop_Model_Stage =
    SELECT < Model creation rules satisfied >
    THEN    Model_Stage_Status:=Finished
    END;
Create_ModelElementA =
    SELECT Model_Stage_Status=Creating
    THEN    < Create a model element A while ensuring model presentation and intra-consistency rules >
    END;
Create_ModelElementB =
    SELECT Model_Stage_Status=Creating
    THEN    < Create a model element B while ensuring model presentation and intra-consistency rules >
    END;
END
```

Fig. 23. General form of the B machine for a Lyra model

To ensure that the models are created according to the Lyra design flow, we introduce the variable *Model_Stage_Status*. When the creation of the corresponding Lyra model starts, the operation **Start_Model_Stage** assigns the value *Creating* to the *Model_Stage_Status* and this in turn enables creating of model elements. Let us observe that *Model_Stage_Status*=*Creating* is the guard of the **Create_ModelElementA** and **Create_ModelElementB** operations in Fig. 23. When a particular model is created, *Model_Stage_Status* variable is assigned the value *Finished*. This, in turn, triggers creating a subsequent model. The order in which the models are created is orchestrated by the corresponding top machine. Its general form is shown in Fig. 24.

Observe that each machine corresponding to the subsequent model **EXTENDS** the machine for the last created model in that stage and hence all the machines for previously created models. In this way we obtain the top machine for a specific stage by incorporating machines for Lyra models created in that stage.

The B extension mechanism allows us to simulate the order in which Lyra models are created. Namely, after one model is created, the top machine defines which model is to be created next. For instance, if *Model0* should be created first and then *Model1*, the guard of the **Create_Model1_Stage** operation of the top machine has the following form:

$$Model0\_Stage\_Status=Finished \wedge Model1\_Stage\_Status=Empty$$

where the value *Empty* assigned to the variable *Model1_Stage_Status* denotes that creating of the *Model1* has not started yet. Since *Model0_Stage_Status=Finished*, i.e., *Model0* is created, the top machine triggers creating *Model1* by calling the operation **Start_Model1_Stage** from the body of the operation **Create_Model1_Stage**.

```
MACHINE   Stage
EXTENDS   Model1_Stage
INVARIANT
/* intra-consistency rules */
          /* Model0 */
          (Model0_Stage_Status=Finished ⇒ ...)
          /* Model1 */
          (Model1_Stage_Status=Finished ⇒ ...)  ...
OPERATIONS
Create_Model0_Stage =
   SELECT
      Model0_Stage_Status=Empty
   THEN
      Start_Model0_Stage
   END;
Create_Model1_Stage =
   SELECT
      Model0_Stage_Status=Finished ∧ Model1_Stage_Status=Empty
   THEN
      Start_Model1_Stage
   END
...
END
```

Fig. 24. General form of the B machine for a specific Lyra stage

Since we assume that the Lyra models are checked for consistency only after they are created, the invariant of the machine corresponding to a certain Lyra stage guarantees that the intra-consistency rules for a particular model are satisfied only when *Model_Stage_Status=Finished.*

To verify the intra-consistency rules, we should prove correctness of the defined top machines and the machines representing the corresponding Lyra models. This task is facilitated by an automatic tool support available for the B Method – AtelierB [20]. AtelierB generates the required proof obligations and attempts to discharge them automatically. In some cases it requires user's assistance for doing this. Upon discharging all proof obligations the verification process completes. It ensures that all the model elements and models themselves at a specific Lyra stage are created according to the specified structural and intra-consistency rules.

Ensuring inter-consistency of Lyra models in B. To verify inter-consistency, we should ensure that the models at different development stages are not contradictory with each other. In this paper we propose refinement [22] as a technique for checking model inter-consistency. A graphical representation of the proposed approach is given in Fig. 25.

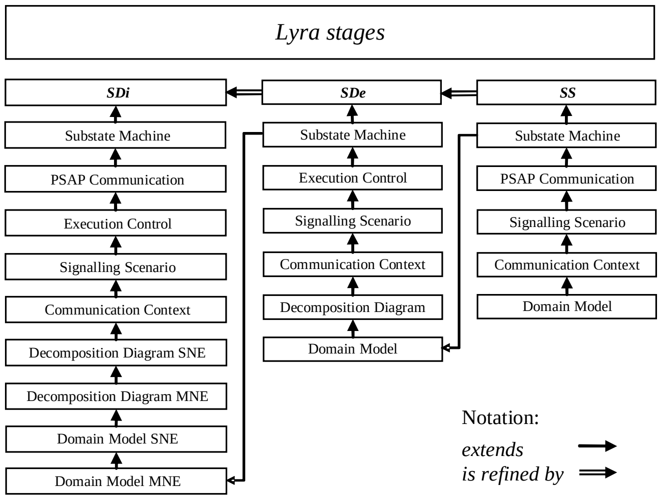


Fig. 25. Overall Lyra development in B

The models in each Lyra stage correspond to the B machines specified according to the pattern given in Fig. 23. The rules of intra-consistency remain unchanged through stages. However, the models starting from the second Lyra stage should be consistent with the models from the previous stages. Hence, we define a B machine corresponding to the top machine of the subsequent Lyra stage as a refinement of the top machine for the previous Lyra stage. Its general form is shown in Fig. 26.

```
REFINEMENT      Stage'
REFINES         Stage
EXTENDS         Model0_Stage'
INVARIANT
/* intra-consistency rules */
            ...
/* inter-consistency rules */
            /* Model0 */
            (Model0_Stage'_Status=Finished ⇒ ... )
            /* Model1 */
            (Model1_Stage'_Status=Finished ⇒... )  ...
OPERATIONS
Create_Model0_Stage =...
Create_Model1_Stage =...
Create_Model0_Stage' =...
Create_Model1_Stage' =...
...
END
```

Fig. 26. General form of the B refinement for the subsequent Lyra stage

The top machine *Stage'* is a superposition refinement of the machine *Stage*. Namely, the existing data structure introduced in the machine *Stage* is unchanged. However, we introduce new variables for the models of the subsequent stage and operations over them. The inter-consistency rules defining the relationships between the model elements from these two stages are expressed as the linking invariant of the refinement *Stage'*. In addition, the invariant of the machine *Stage'* expresses the intra-consistency rules in a similar way as the invariant of the machine *Stage* (see Fig. 24).

Verification of inter-consistency is achieved in a similar way as for intra-consistency. Using Atelier B, we prove correctness of the defined abstract machines corresponding to the models of the subsequent Lyra stage. In addition, we prove that the top machine representing the current Lyra stage is refinement of the top machine representing the previous Lyra stage.

Next we present detailed formal definition of several Lyra models and verification of their consistency.

7.1. Translating Lyra models in B – an example

We start from defining a B machine to represent Domain Model at the SS stage (Fig. 20a). Domain Model is the first Lyra model. Hence, we should ensure only its structural consistency.

While constructing the B machine for Domain Model (*DomainModel_SS* in Fig. 27), we define only the model presentation rules for its elements: *Actor*, *UseCase*, *Association* and *System*. For instance, one of the rules common for all elements in Lyra models postulates that each model element is strictly identified by its unique identifier. This is enforced by typing the introduced variables using the set of unique identifiers (*UNIQUE_ID*).

```
MACHINE      DomainModel_SS
VARIABLES
   Actor, Actor_Name, UseCase, UseCase, Name, System, System_Contains, System_Name, Association, Association_Ends, ...
   DomainModel_SS_Status
INVARIANT
   Actor ⊆ UNIQUE_ID ∧ Actor_Name ∈ Actor ⤔ NAMES ∧
   UseCase ⊆ UNIQUE_ID ∧ UseCase_Name ∈ UseCase ⤔ NAMES ∧
   Association ⊆ UNIQUE_ID ∧ Association_Ends ∈ Association ⤔ (Actor × UseCase) ∧ ...
INITIALISATION
   Actor, Actor_Name  := ∅, ∅ || ... || DomainModel_SS_Status := Empty
OPERATIONS
Start_DomainModel_SS =...
Stop_DomainModel_SS =...
Create_System =...
Create_Actor =
   SELECT DomainModel_SS_Status=Creating
   THEN
      ANY name WHERE name ∈ NAMES ∧ Name_Not_In_Use
      THEN
         ANY idx WHERE idx ∈ UNIQUE_ID ∧ ID_Not_In_Use
         THEN
            Actor := Actor ∪ { idx } || Actor_Name := Actor_Name ∪ { idx ↦ name } || ...
         END
      END
   END;
Create_UseCase =...
Create_Association =...
END
```

Fig. 27. Excerpt from the *DomainModel_SS* machine

The additional model presentation rules are derived from the requirements for Domain Model in the SS stage. For instance, a model presentation rule for the element Actor in Domain Model at the SS stage expresses that an actor has to have the name. Observe that the operation **Create_Actor** in *Domain_Model_SS* machine enforces this rule while creating an actor. Namely, the variable *Actor_Name* contains the names for each created actor. We omitted the detailed presentation of all the operations of *Domain_Model_SS*. They follow the general form of the operations given in Fig. 23.

The next step in Lyra development is creating the Communication Context model (Fig. 20b). To ensure intra-consistency, the machine *CommunicationContext_SS* (Fig. 28) refers to *DomainModel_SS* in its **EXTENDS** clause. The elements of the Communication Context model are the variables of the *CommunicationContext_SS* machine. They are defined relying on the definitions of *DomainModel_SS* machine. The dependencies between the models are formulated as the intra-consistency rules. They implement the requirements obtained for the Communication Context model at SS stage. For instance, an intra-consistency rule for the active classes in Communication Context at the SDe stage states that an active class should be defined for each use case in Domain Model with the same name as the corresponding use case. This rule is specified while creating the element *ActiveClass* in the *CommunicationContext_SS* machine.

```
MACHINE     CommunicationContext_SS
EXTENDS     DomainModel_SS
VARIABLES
   ActiveClass , ActiveClass_Name ,
   ExternalClass , ExternalClass_Name ,
   PSAP_Port , USAP_Port ,
   Interface_IN , Interface_OUT, ...
   CommunicationContext_SS_Status
INVARIANT
   ActiveClass ⊆ UNIQUE_ID ∧ ActiveClass_Name ∈ ActiveClass ⤔ (System ∪ UseCase) ∧ ...
INITIALISATION
    ActiveClass, ActiveClass_Name := ∅, ∅ || ... || CommunicationContext_SS_Status := Empty
OPERATIONS
Start_CommunicationContext_SS =...
Stop_CommunicationContext_SS =
   SELECT ran (ActiveClass_Name) = (UseCase ∪ System) ∧...
   THEN
         CommunicationContext_SS_Status:=Finished
   END;
Create_ActiveClass_For_UseCase =
   SELECT
      CommunicationContext_SS_Status=Creating
   THEN
      ANY id1, idx WHERE id1 ∈ UNIQUE_ID ∧ id1 ∈ UseCase ∧ id1  ∉ ran ( ActiveClass_Name ) ∧
                         idx ∈ UNIQUE_ID ∧ ID_Not_In_Use
      THEN
        ActiveClass := ActiveClass ∪ { idx } || ActiveClass_Name := ActiveClass_Name ∪ { idx ↦ id1 } || ...
      END
   END;
Create_ActiveClass_For_System =...
Create_ExternalClass =...
Create_USAP_Port =...
Create_PSAP_Port =...
END
```

Fig. 28. Excerpt from the *CommunicationContext_SS* machine

The **Create_ActiveClass_For_UseCase** operation (see Fig. 28) creates an active class with the same name as the use case with the unique ID. The guard of the operation **Stop_CommunicationContext_SS** ensures that this model is properly created only when there exists an active class in Communication Context for each use case in Domain Model.

We omit presenting the B machines for Signalling Scenario, PSAP Communication and Substate Machine in the SS stage since they follow the same general form given in Fig. 23. The top machine for the SS stage is obtained according to the pattern shown in Fig. 24.

The inter-consistency rules guide defining Domain Model (Fig. 21a) in the SDe stage. The SDe stage adds new actors to Domain Model. They should be associated with already existing use cases. The machine *DomainModel_SDe* (see Fig. 29) has similar structure as *DomainModel_SS* (see Fig. 27).

```
MACHINE     DomainModel_SDe
EXTENDS     SubstateMachine_SS
VARIABLES
   Actor, Actor_Name1, Association1, Association_Ends1, DomainModel_SDe_Status
INVARIANT
   Actor1 ⊆ UNIQUE_ID ∧ Actor_Name1 ∈ Actor1 ⤔ NAMES ∧
   Association1 ⊆ UNIQUE_ID ∧ Association_Ends1 ∈ Association1 ⤔ (Actor1×UseCase) ∧ ...
INITIALISATION
   Actor1, Actor_Name1 := ∅, ∅ || ... || DomainModel_SDe_Status := Empty
OPERATIONS
Start_DomainModel_SDe =...
Stop_DomainModel_SDe =...
Create_Actor1 =...
Create_Association1 =
   SELECT DomainModel_SDe_Status=Creating
   THEN
      ANY id1,id2,idx
      WHERE id1∈UNIQUE_ID ∧ id1∈Actor1 ∧ id2∈UNIQUE_ID ∧ id2∈UseCase ∧ (id1,id2) ∉ ran (Association_Ends1) ∧
               idx ∈ UNIQUE_ID ∧ ID_Not_In_Use
      THEN
         Association1 := Association1 ∪ { idx } || Association_Ends1 := Association_Ends1 ∪ { idx ↦ (id1,id2) } || ...
      END
   END
END
```

Fig. 29. Excerpt from the *DomainModel_SDe* machine

However, the new variables: *Actor1*, *Actor_Name1*, *Association1* and *Association_Ends1*, are introduced to model the newly introduced elements. Observe that the operation **Create_Association1** enforces the inter-consistency rule: it represents the associations between the variable *UseCase* from the *DomainModel_SS* and the introduced variable *Actor1* in *DomainModel_SDe*.

B development in the SDe stage proceeds as shown in Fig. 25 and finishes with defining the refinement *SDe* (Fig. 30), which is obtained using the pattern given in Fig. 26.

```
REFINEMENT  SDe
REFINES     SS
EXTENDS     SubstateMachine_SDe
INVARIANT
/* intra-consistency rules */
    ...
/* inter-consistency rules */
   /* Domain Model */
   (DomainModel_SDe_Status=Finished ⇒ ran(Association_Ends1)⊆(Actor1×UseCase)) ∧ ...
  /* Decomposition Diagram */
   (DecompositionDiagram_SDe_Status=Finished ⇒ (Association_Source2[dom(Association_Target2)]=UseCase)) ∧ ...
OPERATIONS
Create_DomainModel_SS =...
Create_CommunicationContext_SS =...
Create_SignallingScenario_SS =...
Create_PSAPComm_SS =...
Create_SubstateMachine_SS =...
Create_Domain_Model_SDe =
   SELECT
      DomainModel_SDe_Status=Empty ∧ PSAPCommunication_SS_Status=Finished
   THEN
      Start_DomainModel_SDe
   END;
Create_DecompositionDiagram_SDe = ...
Create_CommunicationContext_SDe =...
Create_SignallingScenario_SDe = ...
Create_ExecutionControl_SDe = ...
Create_SubstateMachine_SDe =...
END
```

Fig. 30. Excerpt from the *SDe* refinement

The invariant of the refinement *SDe* expresses not only the intra-consistency rules addressed at the SDe stage but also the inter-consistency rules between models on SS and SDe stages. For instance, for Domain Model in SDe stage to be consistent with Domain Model in SS stage, it should associate the newly added *Actor1* with *UseCase* from the same model in the SS stage, i.e., $\mathbf{ran}(Association\_Ends1) \subseteq (Actor1 \times UseCase)$ should hold. By proving refinement between the corresponding top machines, we verify inter-consistency of Lyra models from the SS and SDe stages.

The SDi stage is handled in the similar way, resulting in a set of B machines for the corresponding Lyra models and a refinement *SDi* – a top machine for this stage. A graphical representation given in Fig. 25 summarizes the overall process of Lyra formalization, allowing us to establish consistency among models in the Lyra development flow.

8. Related work

There are several formal approaches to ensuring consistency of UML models. Engels et al. describe in [23] how to formalize the consistency of models in UML-RT – a dialect of UML for modelling concurrent systems. They focus on translating UML-RT statechart diagrams into CSP and ensuring their consistency during model evolution. Similarly, our approach ensures the consistency between models on different development stages via refinement in B. However, we consider a wider set of UML models.

Van Der Straeten et al. [24, 25] propose an extension of the UML metamodel, namely the UML Profile for Model Consistency, supporting the consistency between different versions of a model. They check consistency by translating the UML Profile into the description logic (DL). Logic rules are then used to detect model inconsistencies. Moreover, this approach does not consider preserving consistency between different levels of abstraction and also uses a limited subset of UML (i.e., only class, sequence and state diagram).

Ensuring intra-consistency of UML models has been addressed by Kim and Carrington [26]. They describe how consistency constraints of UML model elements (i.e., elements of the UML metamodel) can be formally defined at a language level using Object-Z. This formal meta-modelling approach to defining UML modelling concepts is based only on UML State Machine. The metaclasses from UML State Machine are translated into Object-Z classes. Consistency constraints are defined as invariants on these Object-Z classes. Consistency between different UML models is checked via verifying that model elements composing the models preserve all consistency constraints attached to their metaclasses. Therefore, the approach deals with intra-consistency only, while our approach handles both intra- and inter-consistency.

The use of Object-Z to reason about model consistency was also studied by Rash and Wehrheim [27]. They give formal semantics to UML classes and state machines using Object-Z. As a common semantic domain for both classes and state machines, they use semantic model of the process algebra CSP. Consistency checking is then achieved by translating the obtained Object-Z specification into CSP. Similarly to our approach, they use refinement for model evolution and show how consistency is preserved, but on a limited subset of UML.

The problem of consistency has mostly been studied for UML class and state diagrams. Meanwhile, only a few researches considered less formal concepts of UML, such as use cases, sequence and activity diagrams. In [28], Krishnan proposes an approach that defines UML diagrams (including use case diagrams) in terms of state predicates. The consistency between various diagrams is then verified using the theorem prover PVS. Although our approach comprises both formal and informal UML descriptions similarly as [28], it also reasons about consistency during model evolution.

The approaches to consistency of UML models based on their translation to some formal notation are the most common, as observed in [29]. However, there are many approaches [30, 31, 32, 33, 34] which are grouped around the constraint definition languages, in particular OCL, proposing different enhancements of OCL to enable better expressiveness of constraints. Naturally, these approaches show how intra-consistency between UML models can be achieved. However, to the best of our knowledge, there is no research addressing inter-consistency checking using OCL.

9. Conclusion

In this paper we formalized and formally verified Lyra development flow represented as the Lyra profile. This work establishes a basis for automating model-driven development of distributed communicating systems and communication protocols.

We made two technical contributions. The first is the definition of the Lyra UML2 profile for developing communicating systems and protocols conforming to specified architectural rules. The profile has been derived as a result of a number of large industrial developments conducted according to the Lyra methodology within Nokia Research Center. The profile defines the Lyra-specific modelling concepts and dependencies between them, thus outlining the required stages of the system development. The profile is considered to be a reference model using which we could validate created Lyra models. Validation ensures that these models use only concepts defined by the architectural rules. We discussed the related work in this area while presenting the major profiling principles.

The second contribution is specification and verification of the Lyra design method within the formal modelling framework – the B Method. This work allowed us to establish consistency between the Lyra UML2 models while undertaking the Lyra development, which otherwise we could not achieve within the profile solely. While verifying the Lyra development flow, we simulated Lyra development and formalized both the Lyra models and the intra- and inter-consistency rules in B. The Lyra models are translated into the corresponding B machines according to the proposed patterns. The intra-consistency rules are expressed as the invariant of the top machine for each particular stage. The inter-consistency rules are defined as the linking invariant in the refinement machines corresponding to the subsequent stages. Full formal verification of the obtained specifications and refinements is done using an automatic tool support for the B Method – Atelier B. It guarantees both intra- and inter-consistency of models created at various stages of Lyra development.

In general, the presented approach establishes a basis for automating the Lyra design flow. It not only defines a profile supporting the entire development process of communicating systems and communication protocols, but also smoothly integrates formal verification for ensuring model consistency.

As our future work we are planning to extend the proposed approach to define and verify behavioural consistency as well. It would complement the structural consistency we have defined and presented in this paper.

References

[1] B. Selic, “The Pragmatics of Model-Driven Development,” IEEE Software, Volume 20, Issue 5, 2003, pp. 19 – 25.

[2] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[3] OMG. (2006, March). UML 2.0 Infrastructure Specification. [Online]. Available: http://www.omg.org/docs/formal/05-07-05.pdf

[4] OMG. (2005, August). UML 2.0 Superstructure Specification. [Online]. Available: http://www.omg.org/docs/formal/05-07-04.pdf

[5] S. Leppänen, M. Turunen, and I. Oliver, “Application Driven Methodology for Development of Communicating Systems,” presented at the Forum on Specification and Design Languages, Lille, France, September 2004.

[6] S. Leppänen, "The Lyra Method," Tampere University of Technology, Finland, Technical Report, 2005.

[7] J.-R. Abrial, *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[8] J. Derrick, D. Akehurst, and E. Boiten, "A framework for UML consistency," in *Proceedings of the <<UML>> 2002 Workshop on Consistency Problems in UML-based Software Development*, 2002, pp. 30-45.

[9] OMG. (2006, May). Object Constraint Language (OCL) 2.0 Specification. [Online]. Available: http://www.omg.org/docs/formal/06-05-01.pdf

[10] OMG. (2002, April). UML Profile for CORBA. [Online]. Available: http://www.omg.org/docs/formal/02-04-01.pdf

[11] OMG. (2005, January). UML Profile for Schedulability, Performance, and Time Specification. [Online]. Available: http://www.omg.org/docs/formal/05-01-02.pdf

[12] OMG. (2006, May). UML Profile for Modeling Quality of Service (QoS) and Fault Tolerance Characteristics and Mechanisms. [Online]. Available: http://www.omg.org/docs/formal/06-05-02.pdf

[13] 3GPP Organizational Partners. (2006, June). Technical specification 25.305: Stage 2 functional specification of UE positioning in UTRAN. France. [Online]. Available: http://www.3gpp.org/ftp/Specs/archive/25_series/25.305/25305-730.zip

[14] P. Selonen and J. Xu, "Validating UML Models against Architectural Profiles," in *Proceedings of the 9th European Software Engineering Conference (ESEC'03)*, 2003, pp. 58-67.

[15] P. Selonen, *Model Processing Operations for the Unified Modeling Language*. Doctoral dissertation, Tampere University of Technology, Finland, 2005.

[16] C. Riva, P. Selonen, T. Systä, and J. Xu, "UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, 2004, pp. 50-59.

[17] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille, "Consistency Problems in UML-Based Software Development," in *UML Modeling Languages and Applications*, LNCS 3297, Springer-Verlag, 2005, pp. 1-12.

[18] S. Schneider, *The B Method. An introduction*. Palgrave, 2001.

[19] MATISSE Handbook for Correct Systems Construction. (2003). EU-project MATISSE: Methodologie and Technologies for Industrial Strength Systems Engineering, IST-199-11345. [Online]. Available: http://www.esil.univ-mrs.fr/~spc/matisse/Handbook

[20] ClearSy, *Atelier B - User Manual (Version 3.6)*, Aix-en-Provence, France, 2003.

[21] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall International, 1976.

[22] R. J. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

[23] G. Engels, J. M. Kuster, R. Heckel, and L. Groenewegen, "Towards Consistency–Preserving Model Evolution," in *Proceedings of the International Workshop on Principles of Software Evolution*, 2002, pp. 129-132.

[24] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers, "Using Description Logic to Maintain Consistency between UML Models," in *UML 2003*, LNCS 2863, 2003, pp. 326-340.

[25] J. Simmonds, R. Van Der Straeten, V. Jonckers, and T. Mens, "Maintaining consistency between UML models using description logic," L' Objet (Objet) Vol. 10, LMO'04, 2004, pp. 231-244.

[26] S.-K. Kim and D. Carrington, "A Formal Object-Oriented Approach to defining Consistency Constraints for UML Models," in *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, 2004, pp. 87-94.

[27] H. Rasch and H. Wehrheim, "Checking Consistency in UML Diagrams: Classes and State Machines," in *FMOODS 2003*, LNCS 2884, 2003, pp. 229-243.

[28] P. Krishnan, "Consistency Checks for UML", in *Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC'00)*, 2000, pp. 162-169.

[29] M. Elaasar and L. Briand, "An Overview of UML Consistency Management," Carleton University, Canada, Technical Report SCE-04-18, August 2004.

[30] J.-L. Sourrouille and G. Caplat, "Checking UML Model Consistency," in *Proceedings of the <<UML>> 2002 Workshop on Consistency Problems in UML-based Software Development*, 2002, pp. 1-15.

[31] D. Chiorean, M. Pasca, A. Cârcu, C. Botiza, S. Moldovan, "Ensuring UML Models Consistency Using the OCL Environment," Electronic Notes in Theoretical Computer Science, Volume 102, 2004, pp. 99-110.

[32] H. Gomaa and D. Wijesekera, "Consistency in Multiple-View UML Models: A Case Study," in *Proceedings of the <<UML>> 2003 Workshop on Consistency Problems in UML-based Software Development II*, 2003, pp. 1-8.

[33] R. Wagner, H. Giese, and U. Nickel, "A Plug-In for Flexible and Incremental Consistency Management," in *Proceedings of the <<UML>> 2003 Workshop on Consistency Problems in UML-based Software Development II*, 2003, pp. 78-85.

[34] J.-P. Bodeveix, T. Millan, C. Percebois, C. Le Camus, P. Bazex, and L. Feraud, "Extending OCL for verifying UML models consistency," in *Proceedings of the <<UML>> 2002 Workshop on Consistency Problems in UML-based Software Development*, 2002, pp. 75-90.

ISBN 978-952-12-1919-1
ISSN 1239-1891