TUCS

Dubravka Ilić | Elena Troubitsyna |
Linas Laibinis | Sari Leppänen

Formal Verification of Consistency in Model-Driven Development of Distributed Communicating Systems and Communication Protocols

Turku Centre *for* Computer Science

TUCS Technical Report
No 777, June 2006

Formal Verification of Consistency in Model-Driven Development of Distributed Communicating Systems and Communication Protocols

Dubravka Ilić

Elena Troubitsyna

Linas Laibinis

Åbo Akademi University, Department of Information Technologies
Lemminkäisenkatu 14A, 20520 Turku, FIN

Sari Leppänen

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

TUCS Technical Report

No 777, June 2006

Abstract

Currently UML2 is widely used for modelling software-intensive systems. Model driven development of complex software typically starts from abstract, high-level UML2 models which specify the system from several different viewpoints. Abstract models are further refined into more detailed design models in successive development stages. While specifying various aspects and abstraction levels of such systems, we create a set of different models, which should be inter- and intra-consistent. In this paper we propose an approach to ensuring consistency in Lyra – a rigorous, service-oriented and model-based method for developing industrial telecommunication systems and communication protocols. We derive informal requirements to ensuring intra- and inter-consistency and then formalize them in the B Method. The formalization in B allows us to structure complex informal requirements and formally ensure intra- and inter-consistency of models created at various stages of the Lyra development.

Keywords: consistency of UML2 models, intra-consistency, inter-consistency, the B Method, refinement

TUCS Laboratory
Distributed Systems Design

1. Introduction

Recently various model-driven approaches have emerged to support design-centric software development. They promote system development by gradual transformation of system models expressed in Unified Modelling Language (UML) [1]. Modelling typically starts from abstract, high-level models which are iteratively transformed into more detailed design models. However, even at an abstract level a system can be described from different viewpoints. A created set of models becomes even larger at further development stages. To ensure correctness of the developed system, we need techniques for managing model consistency. 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 [2, 3]. Lyra is a model-driven and component-based design method for the development of communicating systems and communication protocols. It consists of four consecutive development stages that support systematic refinement of the design models. The constructed models define externally observable behaviour of system-level services. Lyra has been developed at Nokia Research Center and applied in large-scale UML2-based industrial software development projects.

In this paper, we derive general patterns of UML2 models created at different stages of Lyra development and express intra- and inter-consistency rules for them. Then we define the rules as formal specifications in the B Method [4]. Hence the B Method serves as a common semantics for UML2 models. Our approach to ensuring consistency of UML2 models is similar to the approach based on defining a common semantics of UML presented in [5]. Formal verification of obtained B models 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 introduce the Lyra design method, describe the UML2 models used in the design, and define dependencies between them. Section 3 gives a short introduction to our modelling framework – the B Method. In Section 4 we describe our approach to ensuring intra- and inter-consistency in Lyra by formal specification and refinement in B. Section 5 summarizes the proposed approach and outlines future work.

2. Overview of Lyra Design Method

Lyra [2, 3] 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 system development projects.

Lyra has four main stages: *Service Specification, Service Decomposition, Service Distribution and Service Implementation.* The *Service Specification (SS)* stage focuses on defining the services provided by the system and the different types of users of these services. 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 at the previous stage is decomposed into a set of service components and logical interfaces between them. This stage yields the logical architecture of the system services. In *Service Distribution (SDi)* stage the logical architecture of services is distributed over a given platform architecture. Finally, in *Service Implementation* stage the structural elements are integrated into the target environment. This results in a model which can be used, e.g., as a source for automatic code generation. A detailed description of the Lyra Method can be found in [2, 3].

Lyra uses UML2 [6] as a modelling language. At each Lyra stage we define a set of UML2 models. The models specify the system under construction from the various viewpoints. Moreover, the system is developed in a top-down fashion, hence the models at each subsequent stage represent the system at lower level of abstraction. While developing a system we should ensure model consistency, i.e., 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., structural requirements imposed on the modelling elements. On the one hand, a model has to be consistent with the models at the same development stage. On the other hand, it should be consistent with models at 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*; whereas the *inter-consistency* is defined as the consistency among modelling concepts from different development stages.

To illustrate Lyra development flow and present consistency rules, next we give an example – an excerpt from the development of the 3GPP[1] positioning system. The system provides the positioning service for calculating the physical location of a given user equipment in a mobile network. A complete set of informal specifications of the service can be found elsewhere (e.g., [7]).

Models at Service Specification stage. The system development starts with creating the Domain Model. The Domain Model is a UML2 use case model describing the system services and their users. Its general form is given in Fig. 3a. The Domain Model for the 3GPP positioning system is shown in Fig. 1.

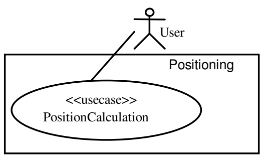

Fig. 1. Domain Model of the Positioning System at the SS stage

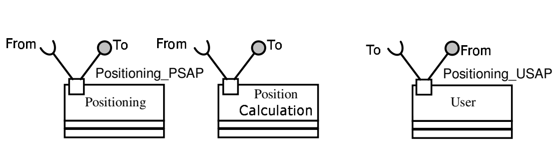

Fig. 2. Communication Context of the Positioning System at the SS stage

[1] Third Generation Partnership Project

To be properly defined, the Domain Model should satisfy certain structural requirements, e.g., an association can associate only an already created actor and a use case. In our example they are User and PositionCalculation correspondingly.

From the Domain Model we derive formal system structure – a UML2 class diagram – called the Communication Context. The general form of this model is shown in Fig. 3b. The Communication Context of the Positioning System is depicted in Fig. 2. To be consistent with the previously created Domain Model, the Communication Context should satisfy a number of intra-consistency rules. For example,

- the Communication Context has an active class for each use case in the Domain Model and the system itself. These are the active classes Positioning and PositionCalculation in the Communication Context in Fig. 2.
- the Communication Context defines an external class for each actor of the Domain Model (the class User in Fig. 2).
- for each active class in the Communication Context we define Provided Service Access Points (PSAPs). Each association connecting an actor and a use case in the Domain Model corresponds to a PSAP. PSAPs are UML2 ports (see Positioning_PSAP and PositionCalculation_PSAP in Fig. 2).
- for each external class in the Communication Context we define Used Service Access Points (USAPs). Each association connecting an actor and a use case in the Domain Model corresponds to a USAP. USAPs are UML2 ports as well (see Positioning_USAP in Fig 2).

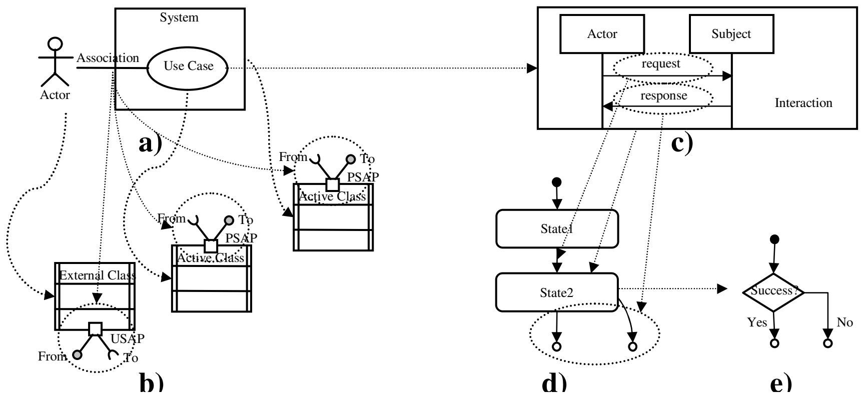

Fig. 3. The design flow of the SS stage

The next model at the SS stage – the Signalling Scenario (Fig. 3c) – is a UML2 sequence diagram, which gives an informal description of the communication between the system service and its user(s). The communication is defined in terms of interactions. Each interaction is a set of Signalling Scenarios defined for a particular system service.

Formally, the communication between the system service and its users is expressed in the PSAP Communication model (Fig. 3d), which is a UML2 state machine. Its states are obtained from the interaction defined in the Signalling Scenario. Transitions between the states specify the communications described in the Signalling Scenarios for a particular use case.

The states in the PSAP Communication model are composite. The dynamic behaviour of the service on the level of sub-states composing a state in the PSAP Communication model is defined in the Substate Machines (Fig. 3e). At the SS stage, the Substate Machines non-deterministically model success or failure of service execution.

Models at Service Decomposition stage. To implement its own services, the system usually uses services provided by the external service providers. Their explicit representation is introduced into the system model at the SDe stage. Namely, they are represented as new actors associated with system services in the Domain Model (Fig. 5a). In our example, to provide a position calculation service, at first the 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 Algorithm to produce the final estimation on the UE location. These external service providers – DB, UE, RefLMU and Algorithm – are introduced in the Domain Model created at the SS stage as the corresponding actors associated with the PositionCalculation use case, as shown in Fig.4. To ensure that the Domain Model at the SDe stage does not contradict with the Domain Model at the SS stage, we should also guarantee that the other elements of the model remain unchanged. This is an example of an inter-consistency rule.

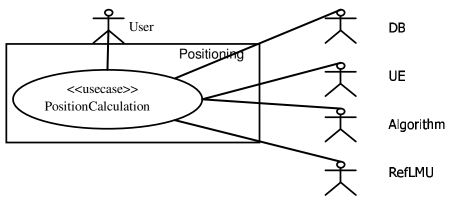

Fig. 4. Domain Model of the Positioning System at the SDe stage

At the SDe stage the intra-consistency rules remain the same as for the SS stage. For instance, while creating the Communication Context (Fig. 5c), we should define external UML2 classes for the actors introduced in the Domain Model at the current stage. Each external class obtains its own PSAP port describing the communication with the system service. Moreover, each association between a system service and an external service provider is modelled as a USAP attached to the already existing active classes.

The decomposition of the system service into sub-services is depicted in the Decomposition Diagram (Fig. 5b). This is an additional model appearing at the SDe stage. The Decomposition Diagram is actually a use-case model showing the sub-use cases that should be executed to provide the system service.

We augment the Signalling Scenario created at SS stage by adding interaction references (ref) representing a set of Signalling Scenarios (Fig. 5e) for each sub-use case. These scenarios describe the communication between the system sub-service and the external service provider. The sub-service execution order is defined by the order in which the references appear in the augmented Signalling Scenario (Fig. 5d).

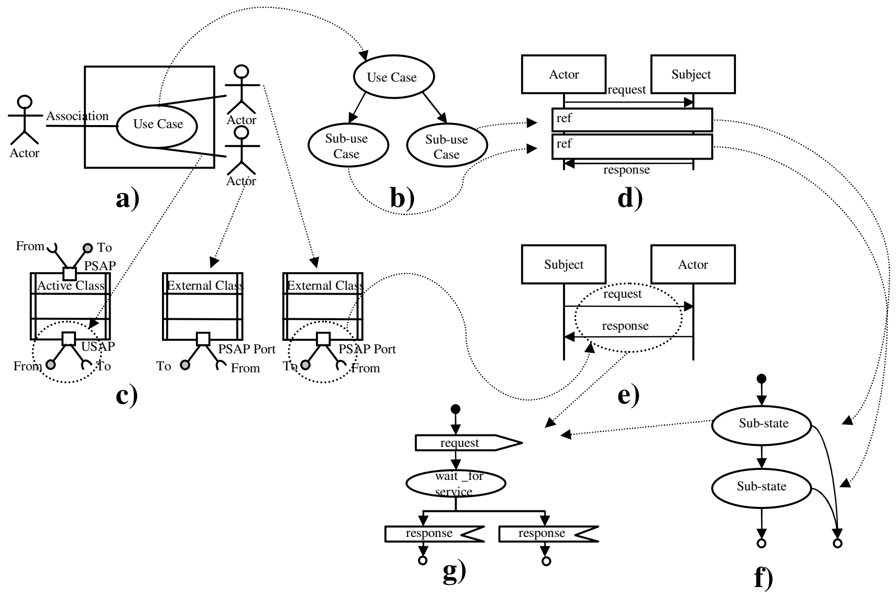

Fig. 5. The design flow of the SDe stage

At the SDe stage the PSAP Communication model is refined to explicitly model the dynamic behaviour on the level of sub-services. The state modelling the actual service execution in the PSAP Communication model is decomposed into a set of sub-states, which are depicted in the Execution Control (Fig. 5f) state machine. The sub-states of the Execution Control state machine correspond to the sub-services. The transitions between the sub-states preserve the order of the interaction references in the Signalling Scenario.

For each sub-state from the Execution Control a Substate Machine (Fig. 5g) should be defined. It models the internal computation and communication between the sub-services.

Models at Service Distribution stage. The SDi stage focuses on distributing decomposed system services over a given platform architecture. The elements of the 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 which communicates with the user is called the Main Network Element (MNE), while the other network elements are called Secondary Network Elements (SNE). The Domain Model at SDi stage should be defined for each of the network elements from its own viewpoint. For instance, when defining the Domain Model for the MNE (Fig. 7a), the rest of the network elements are represented as actors. Similarly, when defining the Domain Model for SNE (Fig. 7b), the MNE and the other existing SNEs are represented as actors. The distribution of the 3GPP positioning system is depicted in Fig. 6. The platform architecture consists of two network elements: Positioning_RNC[2] which is the MNE and Positioning_SAS[3] which is the SNE. The PositionCalculation service distributed over these network elements is represented by the domain models for both of them, i.e., Domain Model for Positioning_RNC and Domain Model for Positioning_SAS as shown in Fig. 6a and 6b respectively. Observe that the Positioning_SAS becomes an actor when presenting the service distribution over the Positioning_RNC network element. Similarly this holds for the actor Positioning_RNC in Fig. 6b. Moreover, the external service providers are also distributed over network elements Positioning_RNC and Positioning_SAS.

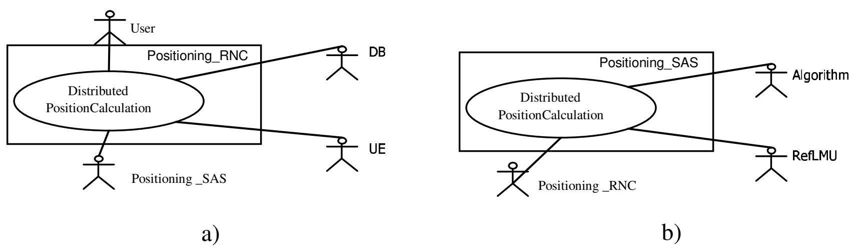

Fig. 6. Domain Model of the Positioning System at the SDi stage

The Communication Context (Fig. 7c) follows the service distribution represented in the domain models by defining active classes for all the distributed services and network elements upon which they are distributed. The external classes defined at the previous Lyra stage remain unchanged. The associations from the Domain Model define interfaces in the Communication Context. They are attached to the USAP and PSAP ports of the classes corresponding to the network elements. The communication between distributed services is defined via the PEER interfaces attached to the PEER ports on the active classes for corresponding network elements.

Distribution of the decomposed functionality of the system is defined by the Decomposition Diagrams. Since the system services and sub-services may be distributed on different network elements, the Decomposition Diagram has to show the system decomposition from the viewpoint of both of them, i.e., we should create the Decomposition Diagram for the MNE (Fig. 7d) and the Decomposition Diagram for the SNE (Fig. 7e).

The Signalling Scenarios (Fig. 7f) for the distributed services introduce interaction references for distributed sub-use 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, the Substate Machine attached to one of its composite distributed states is replaced with a new Execution Control machine (Fig. 7g) defining the distributed functionality in a remote location. It is defined from the viewpoint of the MNE. Additionally, new PSAP Communication state machine (Fig. 7h) needs to be defined for the distributed service from the viewpoint of the SNE.

[2] Radio Network Controller
[3] Stand-alone Assisted Global Positioning System Serving Mobile Location Center

Composite states in the Execution Control machine are further specified by corresponding Substate Machines (Fig. 7i).

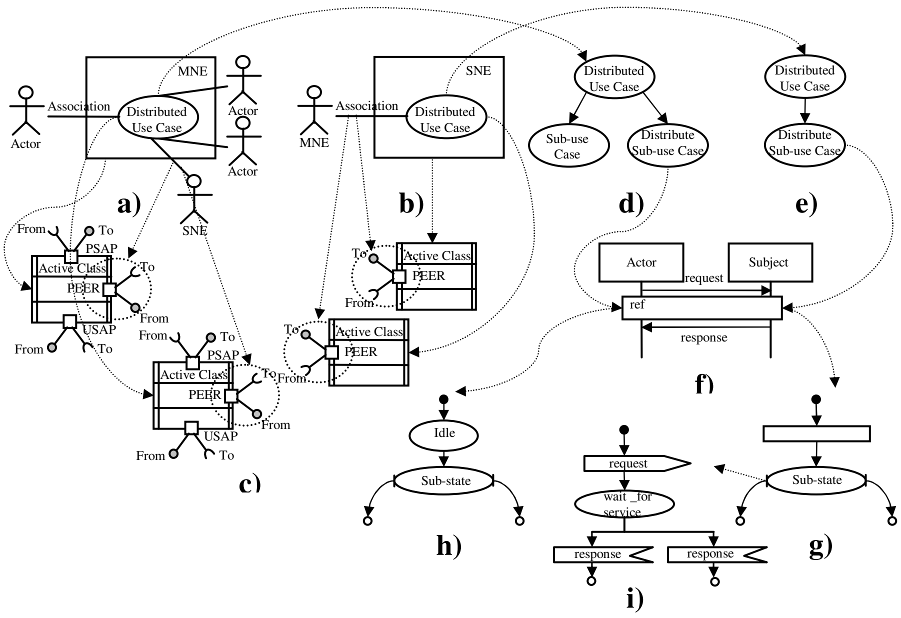

Fig. 7. The design flow of the SDi stage

Fourth Lyra stage – Service Implementation – focuses on implementing low level details on the top of the already existing architecture and does not introduce new consistency constraints. Therefore, we omit its detailed description which can be found elsewhere (e.g., [3]).

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 argue that by formalizing these requirements and the models in Lyra the design process can be automatically verified and required consistency achieved. Next we introduce our framework for formalizing consistency rules – the B Method.

3. The B Method

The B Method [4, 8] 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 [9]. 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 [10], 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 – 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*
INITIALISATION	*Init*
INVARIANT	*I*
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 and they are defined in **OPERATIONS** clause. B statements that we are using to describe the computation in operations have the following syntax:

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

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., [8]).

In this paper we adopt the event-based approach to system modelling [11]. The events are specified as the guarded operations of the form:

Event = **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 also provides structuring mechanisms which enable machines to be expressed as combinations of other machines. Here we use **EXTENDS** clause. When machine M1 extends machine M2, written as **EXTENDS** M2 in the definition of M1, it means that M1 includes M2 and promotes all of the operations of M2, i.e., it provides all of the facilities provided by M2, with some further operations of its own.

To ensure correctness of a B machine, we should verify that the initialization and each operation preserve the invariant and that the invariant is valid, which means that there are some possible machine states which satisfy it.

The formal development in B is based on stepwise refinement [12]. While developing a system by refinement, we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps. 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 more abstract 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. 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 in Lyra models.

4. Formal Verification of Consistency

We start formal verification of consistency of Lyra models by deriving the list of informal requirements. For each Lyra stage we derive the list of requirements corresponding to a particular Lyra model. For each model we group requirements around concrete model elements. Once the complete list of requirements is obtained, we can distinguish between model-presentation, intra-, and inter-consistency rules for particular Lyra models.

The informal requirements form the basis for formalizing Lyra models and consistency rules in B. In general, the approach is as follows: each Lyra model is represented as a B machine of a certain form. Each machine is created in the order defined by Lyra development flow, as described in Section 2. Hence, the set of models defined at each stage is represented by the corresponding set of B machines. The intra-consistency rules are defined as the invariant of a top machine – a machine which includes this set of B machines. The models at each subsequent stage are represented in the same way. Moreover, inter-consistency is ensured by refinement between the corresponding top machines. The refinement relation, defined as a part of the invariant of the top machine, contains inter-consistency rules. Next we present our approach in detail.

Ensuring intra-consistency of Lyra models in B. Ensuring intra-consistency in Lyra requires verifying that the models:

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

To achieve verification of these properties, we first represent each Lyra model as a B machine of a general form given in Fig. 8. The name of the machine corresponds to the name of the 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.

```
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. 8. General form of the B machine for Lyra model

The operations simulate creating of model elements. Namely, for each model element there is one corresponding **Create_ModelElement** operation which allows the creation of the element by enforcing the model presentation and the intra-consistency rules. To ensure that the models are created in a certain order 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 the creation of elements of the model. Observe that *Model_Stage_Status*=*Creating* is the guard of the **Create_ModelElementA** and **Create_ModelElementB** operations in Fig. 8. When a particular model is created, *Model_Stage_Status* variable is assigned value *Finished*. The creation of models at each particular stage is orchestrated by the corresponding top machine. Its general form is shown in Fig. 9. After one model is created, the top machine corresponding to that stage defines which model is to be created next. Namely, if the *Model1* should be created after the *Model0* then the guard of the **Create_Model1_Stage** operation of this 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 the creation of the *Model1* has not started yet. The creation of the *Model1* is then triggered by the operation **Start_Model1_Stage** called from the body of the operation **Create_Model1_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.*

```
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. 9. General form of the B machine for the specific Lyra stage

```
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. 10. General form of the B refinement for the subsequent Lyra stage

To verify the intra-consistency rules, we should prove correctness of the defined top machines and abstract machines representing Lyra models. To achieve this, we use an automatic tool support available for the B Method – AtelierB [10]. 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.

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 [12] as a technique for establishing model inter-consistency. A graphical representation of the proposed approach is given in Fig. 11.

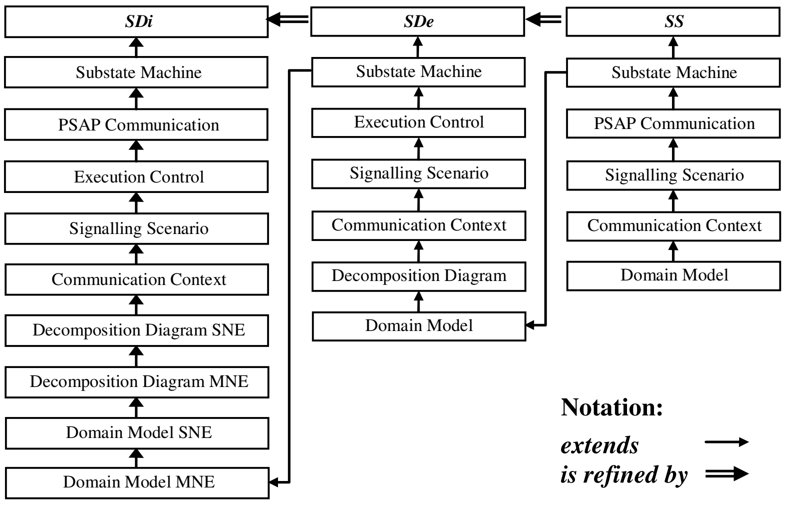

Fig. 11. Overall Lyra development in B

The models from each Lyra stage correspond to the B machines specified according to the pattern given in Fig. 8. The rules of intra-consistency remain unchanged through stages. However, the models starting from the second Lyra stage are obtained based on the models from the previous stage. A B machine corresponding to the top machine of subsequent Lyra stage is a refinement of the top machine for the previous Lyra stage and its general form is shown in Fig. 10.

The top machine *Stage'* uses a specific form of data refinement called superposition refinement [12]. Superposition refinement introduces new variables while leaving the existing data structure unaffected. Observe that the general ideas of superposition refinement and model transformation during the Lyra development process coincide. Each development stage introduces a new set of models, while the models created at the previous stage remain unchanged. The way that elements of the models from one stage relate to the elements from the models in another stage defines the inter-consistency rules between these two stages. These rules are enforced while creating the elements of Lyra models in the subsequent Lyra stages. Although the refinement *Stage'* has a similar form as the machine *Stage* (see Fig. 9), the invariant of the refinement *Stage'* additionally expresses not only the intra- but also inter-consistency rules. The inter-consistency rules are expressed as the linking invariant of the refinement *Stage'*.

To verify the inter-consistency rules, we should prove correctness of defined abstract machines corresponding to the models of the subsequent stage and the refinement of this stage.

We illustrate the process of translating Lyra models into the B machines and their verification, i.e., ensuring model consistency, by an example.

4.1 Translating Lyra models in B – an example

We start our B development by creating the B machines for the models of the SS stage. The first machine to be created is the Domain Model (Fig. 3a). Domain Model is considered as an initial Lyra model and hence has no intra-consistency rules. Therefore, while constructing the B machine for the Domain Model (*DomainModel_SS* in Fig. 12), we define only the model presentation rules for its elements: *Actor*, *UseCase*, *Association* and *System*.

```
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 =...
Create_UseCase =...
Create_Association =...

END
```

Fig. 12. Excerpt from the *DomainModel_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. 13. Excerpt from the *CommunicationContext_SS* machine

These rules postulate that each model element is strictly identified by its unique identifier (*UNIQUE_ID*). Additionally, the model presentation rules are specified from the requirements for the Domain Model in the SS stage. For instance, a model presentation rule for the element Actor in the Domain Model at the SS stage expresses that an actor has to have the name. It is specified as a newly introduced variable *Actor_Name* in Fig. 12. The operations of *Domain_Model_SS* follow the general form given in Fig. 8.

The next step in Lyra development is creating a Communication Context (Fig. 3b) model from the already created Domain Model. Hence, the machine *CommunicationContext_SS* (Fig. 13) refer to *DomainModel_SS* in its **EXTENDS** clause. The elements of the Communication Context model are variables in the *CommunicationContext_SS* machine. They are defined using the variables of *DomainModel_SS* machine. These dependencies 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 element Active Class in the Communication Context at the SDe stage postulates that an active class should be defined for each use case in the 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. The **Create_ActiveClass_For_UseCase** operation creates an active class with the same name as the use case with the unique ID – *id1* for which the active class has not yet been created. The guard of the operation **Stop_CommunicationContext_SS** ensures that this model is properly created only when there exists an active class in the Communication Context for each use case in the Domain Model.

The B machines for the Signalling Scenario, PSAP Communication and Substate Machine in the SS stage also follow the general form given in Fig. 8. Moreover, the machine for the SS stage is obtained according to the pattern shown in Fig. 9.

The inter-consistency rules define how the Domain Model (Fig. 5a) in the SDe stage should be created according to the already created Domain Model in the SS stage. The SDe stage allows new actors to be added in the Domain Model. They can be associated with already existing use cases. Hence, the machine *DomainModel_SDe* (Fig. 14), has similar structure as *DomainModel_SS* (Fig. 12). The new variables: *Actor1*, *Actor_Name1*, *Association1* and *Association_End1*, are introduced to allow modelling of the newly introduced elements. Observe that the operation **Create_Association1** enforces the inter-consistency rule: it allows associations between the variable *UseCase* from the *DomainModel_SS* and the introduced variable *Actor1* in *DomainModel_SDe*.

Further B development in the SDe stage proceeds according to the outline given in Fig. 11 and finishes with defining the refinement *SDe* (Fig. 15), which is obtained using the pattern given in Fig. 10. 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, the Domain Model in SDe stage is consistent with the Domain Model in SS stage if it associates newly added *Actor1* with the *UseCase* from the same model in the SS stage, i.e., if $\mathbf{ran}(Association\_Ends1) \subseteq (Actor1 \times UseCase))$ holds. By establishing refinement, we verify inter-consistency of Lyra models from the SS and SDe stages.

```
MACHINE     DomainModel_SDe
EXTENDS     SubstateMachine_SS
VARIABLES
    Actor1 , 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. 14. Excerpt from the *DomainModel_SDe* machine

```
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. 15. Excerpt form the from the *SDe* refinement

The SDi stage is handled in the similar way. Due to a lack of space we omit a detailed representation of the formal specifications obtained by the refinement process for this stage. However, we give a graphical representation (Fig. 11) which summarizes the overall process of Lyra formalization, allowing us to establish consistency among models in the Lyra development flow. The specification of the full development can be found at: http://www.abo.fi/~dilic/LYRA_spec.

5. Conclusions

In our paper we proposed a formal approach to establishing consistency between UML2 models in the Lyra development method. We showed how to formalize Lyra models in B and express the intra-consistency rules guaranteeing consistency of models in each particular Lyra stage. Lyra models are translated into the corresponding B machines according to the proposed patterns. The intra-consistency rules are enforced in the operations specifying creating of model elements. The rules are collected in the form of the invariant of the top machine. Moreover, we demonstrated how to formally express and verify inter-consistency of the Lyra models created in different stages of B development. Inter-consistency is defined as the linking invariant in the refinement machines corresponding to the subsequent stages. Formal verification of the obtained specifications and refinements is done using an automatic tool support for the B Method – Atelier B.

There are several approaches to ensuring consistency of UML models using formal specifications. Engels et al. describe in [13] 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 the model evolution. Similarly, our approach ensures consistency between models on different development stages via refinement in B. However, we consider a wider set of UML models.

Ensuring both intra- and inter-consistency of UML models in B, makes our approach complementary to the work done in [14], which shows how consistency constraints of UML model elements can be formalized using Object-Z. Nevertheless, they focus only on modelling intra-consistency.

The approach presented in our paper establishes a basis for automating the Lyra design flow. Moreover, derived B models can be seen as a formal specification of a tool for checking consistency of Lyra models.

As future work we are planning to further strengthen the proposed approach to automate the Lyra-based development of communicating systems correct by construction.

Acknowledgment

This work is supported by EU funded research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

References

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

[2] S. Leppänen, M. Turunen, and I. Oliver. *Application Driven Methodology for Development of Communicating Systems.* FDL'04, Forum on Specification and Design Languages, Lille, France, September 2004.

[3] S. Leppänen. *The Lyra Method.* Technical report, Tampere University of Technology, Finland, 2005.

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

[5] J. Derrick, D. Akehurst, and E. Boiten. *A framework for UML consistency.* <<UML>> 2002 Workshop on Consistency Problems in UML-based Software Development, pages 30-45, Dresden, Germany, October 2002.

[6] UML 2.0 Infrastructure Specification: http://www.omg.org/docs/ptc/03-09-15.pdf

[7] 3GPP. Technical specification 25.305: Stage 2 functional specification of UE positioning in UTRAN. See http://www.3gpp.org/ftp/Specs/html-info/25305.htm

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

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

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

[11] J.-R. Abrial. *Event Driven Sequential Program Construction.* 2001. http://www.atelierb.societe.com/ressources/articles/seq.pdf

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

[13] 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, Orlando, Florida, pages 129–132, 2002.

[14] 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), pages 87–94, 2004.

ISBN 952-12-1749-9
ISSN 1239-1891