Specifying UML Profile for Distributed Communicating Systems and Communication Protocols

Sari Leppänen[1], Dubravka Ilic[2], Qaisar Malik[2], Tarja Systä[3] and Elena Troubitsyna[2]

[1] Nokia Research Center, Computing Architectures Laboratory,
P.O. Box 407, 00045, Helsinki, Finland
Sari.Leppanen@nokia.com
[2] TUCS, Åbo Akademi, Department of Computer Science,
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
(Dubravka.Ilic, Qaisar.Malik, Elena.Troubitsyna)@abo.fi
[3] Tampere University of Technology, Institute of Software Systems
P.O. Box 553, FI-33101 Tampere, Finland
Tarja.Systa@tut.fi

Abstract. Usually development of modern software is tackled from several different viewpoints and in a number of iterations. While specifying various aspects and abstraction levels of the system under construction, we create a set of different models, which should be inter- and intra-consistent. Currently UML is widely used for modeling software-intensive systems. To handle consistency better, UML's built-in extension mechanism, *profile*, can be used. In this paper we present a specification of a profile for modeling distributed communication systems and protocols. We identify the general patterns of models created at different stages of Lyra – a rigorous, service-oriented and model-based method for developing industrial telecommunication systems and communicating protocols. The proposed profile with consistency constraints is formalized in the B Method. Formalization in B helps us to ensure intra- and inter-consistency of models created at various phases of Lyra development. Hence it potentially increases our confidence in correctness of developed software.

1 Introduction

Recently various model-driven approaches have emerged to support more architecture and design-centric software development. Modeling typically starts from abstract, high-level models that are refined into the detailed design models in successive development stages. The variety of models, both in terms of view points and levels of abstraction, require techniques for managing model consistency. On the one hand, we need to ensure *intra-consistency* of the models, i.e., consistency among concepts specifying different aspects of the system structure and behaviour on the same development stage. On the other hand, we should establish *inter-consistency* of models,

i.e., demonstrate consistency among modeling concepts from the different development stages.

An acute problem in modeling with UML is to validate design models with respect to architectural rules. In UML2 [1], the architectural rules can be defined in a systematic way using built-in light-weight extension mechanism called *profiles*. The profiles can help the software designers to tackle this problem. Moreover, with the support of a proper tool, the design of models can be automatically checked against the profiles. UML profiles can thus provide a solid basis for increasing the level of automation of software engineering.

In this paper we introduce a specification of a UML2 profile, called *Lyra profile*. Lyra [2] is a model-driven and component-based design method for the development of communicating systems and communication protocols. It consists of four consecutive development phases that support systematic refinement of the design models. The models constructed 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.

We identify the general patterns of UML2 models created at different stages of Lyra method. We define intra- and inter-consistency rules for them. We present an approach to ensuring intra- and inter-consistency of Lyra models via specification and refinement in the B method [3].

The paper is structured as follows: in Section 2 we describe the Lyra design method and give a short introduction to UML2 metamodeling and Lyra profiling principles. In Section 3 we present the Lyra profile and give its graphical representation. In Section 4 we describe an approach to ensure intra- and inter-consistency in Lyra by formal specification and refinement in the B Method. In Section 5 we discuss the proposed approach and outline the future work.

2 The Lyra Method and Lyra Profile in UML2

2.1 Overview of Lyra

Lyra [2, 4] 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. It has been successfully applied in several large-scale industrial system development projects.

Lyra has four main phases: *Service Specification, Service Decomposition, Service Distribution and Service Implementation*. The Service Specification phase focuses on defining the services provided by the system and the different types of users of these services. In this phase the externally observable behaviour of the system level services on the corresponding logical user interfaces is defined. In the Service Decomposition phase the abstract model produced at the previous stage is decomposed in a stepwise and top-down fashion into a set of service components and logical interfaces between them, resulting in the logical architecture of the system level services. In Service Distribution phase the logical architecture of services is distributed over a given platform

architecture. In Service Implementation phase the structural elements are adjusted and integrated to the target environment, resulting in a model, which can be used, e.g., as a source for automatic code generation.

The example of Lyra development can be found in [5, 6].

2.2 UML metamodeling and profiles

The latest UML version 2.0 introduces major changes to the version 1.5. The most significant structural change is the division of the specification in Infrastructure and Superstructure specifications, which respectively define the foundation language constructs and the user-level constructs required for UML2. UML2 Infrastructure is assumed to be extensively reused by various metamodel definitions. For instance, Meta-Object Facility (MOF) reuses it to provide the ability to model metamodels and UML2 Superstructure reuses it to define UML metamodel.

To enable flexible reuse, UML2 infrastructure elements are structured in various packages. When reusing the infrastructure, the packages included in the metamodel should be clearly identified. These packages could be imported as such or via specialization. For instance, UML::Classes::Kernel package imports all the Infrastructure::Core subpackages and extends some of their classes via specialization [1]. Such extensions should naturally be clearly defined. The extensions can also use the same name as the extended class, as long as they have different namespace, namely, they are placed in different packages. For instance, UML2 Superstructure introduces the class *Class* from *Communications* package that generalizes the class *Class* from *Kernel* package and class *BehavioredClassifier* from *BasicBehaviors* package. *Class (from Communications)* is a class that can be designated as active, when each of its instances have their own thread, or passive, when each of its instances execute in the context of some other object. The extension mechanism used means that the unique name of a metamodel element (a class) consists of the name of the element itself and the package it belongs to (e.g., "Class (from Communications)").

UML2 Superstructure defines the UML metamodel itself, (re)using the Infrastructure specification. Namely, UML2 Superstructure defines the user level constructs required for UML2. The two complementary specifications, Infrastructure and Superstructure, constitute a complete specification for the UML2 modeling language [1].

As in UML 1.x, *profiles* are the built-in light-weight extension mechanism of UML2 standard. Profiles can be used to extend a MOF-based metamodel, e.g., UML metamodel, for a specific context, domain or purpose. Profiles are only allowed to contain tag definitions, stereotypes, constraints and data types [1, 7]. In UML2, properties can be attached to the introduced stereotypes. They are marked as attributes inside a class representing a new stereotype. The profile mechanism is defined by Profile package in UML2 Infrastructure.

The profile mechanism is not a first-class extension mechanism of UML and thus does not allow modifications of existing metamodels, as the specification states [8]. This means that the new stereotypes introduced, meta-attributes used, and constraints given cannot contradict with the reference metamodel; it is impossible to take away any of the constraints that apply to a metamodel, but it is possible to add new constraints that are specific to the profile. In short, a reference metamodel is considered

always as a "read only" specification. This implies that the specialized semantics is assumed to not contradict with the semantics of the reference metamodel. This restriction of using UML profile mechanism guarantees, e.g., that any CASE-tools compliant with 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 they are subsets of existing associations in the reference metamodel [1]. Being a subset of an association in a reference metamodel means, according to the above-mentioned profiling principles, that the introduced association can be directly mapped with an association of the same type in the reference metamodel and, e.g., the multiplicity ranges must fall in the range of the corresponding multiplicities of the association in the reference metamodel. Such associations provide a convenient and intuitive way to not only define but also model restrictions and constraint defined for the profile. Further, such associations could also be expressed using OCL, which are allowed 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 [9], for schedulability, time and performance [10], for modeling quality of service and fault tolerance characteristics and mechanisms [11] and for Enterprise Distributed Object Computing (EDOC) [12].

2.3 Lyra profiling principles

The construction of Lyra profile was motivated by the earlier work by Selonen and Xu [13, 14] on defining a profile (more precisely, a profile hierarchy) for capturing architectural rules and constraints relevant for a particular product line platform. In [13, 14], Selonen and Xu introduce a concept of an *architectural profile,* which relies on UML 1.5 profile mechanism. Architectural profiles are extended UML profiles specialized for describing architectural constraints and rules for a given domain [14]. In [15] the architectural profiles are used to support maintenance of a large-scale product platform architecture and real-life product-line products built on top of this platform.

Since UML 1.5 profiles prevent the designer from explicitly constraining the inherited meta-associations among user-defined stereotypes, Selonen and Xu use *extended profiles* to address this shortcoming. Extended profiles contain two parts: a standard UML metamodel part showing the subset of the metamodel that is being extended, and an extension part showing the stereotypes and the inherited meta-associations and other constraints. The extension part describes the allowed relationships between the architectural concepts: the classifiers, the interfaces and the dependencies and realization relationships between them. The actual architecture model validated against the profile must satisfy the constraints implied by the profile, i.e., it is not allowed to have other structures except the ones explicitly described in the set of profiles. Selonen and Xu further characterize in [13, 14] how the extension part can be normalized into a standard UML profile using OCL. This normalization mechanism resembles the two

above-mentioned proposed ways to achieve the effect of new (meta)associations, presented in UML2 Infrastructure specification. Namely, 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. Such visualizations are easy for the software architect to read and understand. In the introduction of Lyra profile (section 3), we use corresponding visualizations as used in architectural profiles.

3 The Lyra Profile

In this chapter the main concepts introduced and used in the Lyra design flow will be summarized in form of the *Lyra profile*. The concepts in the Lyra profile are domain specific and independent of the languages and tools used in modeling.

The Lyra profile presents a basic reference model against which the correctness and completeness of the design models can be checked. Correspondingly, information in the profile can be used in automating the design flow by automatic model template generation. To allow consistency checking, consistency rules for both structure and behaviour will be defined and described in the enhanced Lyra profile as constraints. This is the work in progress in the *Rodin* EU project [16]. Here we will present the basic profile without these extensions.

The profile can also be used in the development of automated model transformations for various purposes. For instance, an abstracted verification or testing including only the relevant aspects with respect to chosen properties (on a set of functionalities and interfaces) could be performed based on a design model. The Lyra profile can also be used as a starting point for profiles of different languages used in the system development process.

The Lyra profile has been described using UML2 language. However, this does not make the profile UML2 specific. UML2 is used as a description method and to bring in the basic concepts. This allows us to avoid unnecessary redefinitions, which would focus the presentation incorrectly and obscure the actual purpose. For the sake of clarity, we use for associations the shorthand notation introduced in [14] and illustrated by an example in Fig. 1.

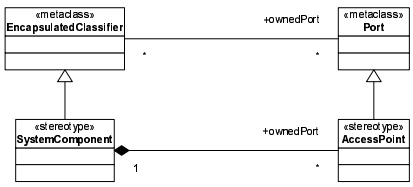

Fig. 1. Shorthand notation for associations

3.1 Structure

System Component - is a structural model element, which encapsulates a logical, independent piece of system specification (and ultimately implementation). A system component may be decomposed into sub-system components, which are system components themselves and may be decomposed further. A SystemComponent can be developed in isolation and later integrated to be a part of a larger system. SystemComponent extends UML2 class *EncapsulatedClassifier (from Ports)*, which extends *StructuredClassifier (from InternalStructures)*. StructuredClassifier extends a classifier with the capability to have internal structure. EncapsulatedClassifier extends a classifier with the ability to own ports. SystemComponent may own several *AccessPoints*, through which it interacts with the environment. SystemComponent may own several *ServiceComponents*, which encapsulate its behavioral specifications.

ServiceComponent - is a logical model element, which encapsulates a set of behavioral specifications. Its total behaviour consists of *ServiceBehavior* and *ServiceComponentBehavior*. ServiceBehavior encapsulates the behavioral specifications related to the provided service. ServiceComponentBehavior encapsulates the behaviour related to implementation specific functionalities. ServiceComponent implements and uses the services characterized by the *Interfaces* it is related to. The services are provided and used through the owned *AccessPoints*. ServiceComponent extends the UML2 *BehavioredClassifier (from BasicBehaviors)*. In UML 1.4 there was no separate metaclass for classifiers with behaviour.

Constraints:

[1] An interface implemented by a ServiceComponent may not be used by the same ServiceComponent.

3.2 AccessPoints

AccessPoint - is a point of communication. An association with role *provided* references the interfaces specifying the set of behavioral features that the owning SystemComponent offers as its services to the environment at this AccessPoint. Correspondingly, the association with role *required* references the interfaces specifying the set of behavioral features that the owning SystemComponent expects to be provided by its environment. AccessPoint is an abstract concept, which cannot be instantiated as such, but through its generalizations *SAP* or *PeerAP*. The base class of AccessPoint is *Port* of UML2.

SAP (Service Access Point) - is a communication point between the system and its environment. Through a SAP a system may either provide its services to external clients or use the services provided by external entities. SAP is an abstract concept, which cannot be instantiated as such, but through its generalizations, i.e., *PSAP* or *USAP*. SAP is a generalization of AccessPoint.

Constraints:

[2] SAP is instantiated either as PSAP or USAP.

PSAP (Provided Service Access Point) - is a communication point between the system and its users. Through PSAP a system provides its services. PSAP encapsulates

communication related to providing a single service or a set of services logically grouped together. PSAP is a generalization of SAP.

Attributes:

isService: true - PSAP is used to publish a service or a set of services to the environment. Service is implemented by the service component the PSAP is attached to.

USAP (Used Service Access Point) - is a communication point between the system and external service providers. USAP encapsulates communication related to using a single service or a set of services logically grouped together. Grouping can be done, e.g., according to communicating peer entity or type of the service used through USAP. USAP is a generalization of SAP.

Attributes:

isService: false - USAP is used to obtain services provided by other entities, allowing it to implement its own services.

PeerAP (Peer Access Point) - is a communication point between a set of distributed model elements. Distributed service components interact through PeerAPs to provide a uniform service (or a set of them) in distributed system architecture. PeerAP is a generalization of AccessPoint.

Attributes:

isService: false - PeerAP is used for communication between distributed service components and is thus part of the internal implementation of a service in a distributed system.

3.3 Behaviour

ServiceComponentBehavior - is invoked when an instance of the owning ServiceComponent is created. ServiceComponentBehavior is not part of the service logics, but merely encapsulates all internal implementation-specific functionalities, like dynamic process management and routing of incoming messages. ServiceComponentBehavior extends the UML2 *Behavior*.

ServiceBehavior - is an abstract behavior specification owned by a ServiceComponent. It represents the composition and the supertype of all behavioral specifications constituting the actual service logics. ServiceBehavior is defined in the context of a ServiceComponent, so it may refer to the features of the owning ServiceComponent. An invoked behavior may invoke other behaviors specified in the same context. This defines how the behavioral specifications interact. As an abstract concept, ServiceBehavior cannot be instantiated as such but through its generalizations *InternalBehavior* and *AccessPointCommunication*. These are the main categories, or types, of ServiceBehavior and correspond to notion of internal and externally observable behavior. ServiceBehavior extends the UML2 *Behavior*.

InternalBehavior - is a generalization and one of the main types of ServiceBehavior. As an abstract concept it can be instantiated only through its generalizations.

ExecutionControl - is of type InternalBehavior and encapsulates behavioral specification defining an execution flow at a certain abstraction level. According to that the other behavioral specifications of a ServiceComponent are invoked. It may not con-

tain other events or actions than *exitPoints* (returned by the invoked behaviors) as preconditions for transitions preceding the execution flow.

InternalComputation - is part of the internal behavior, not visible to the environment of the owning ServiceComponent or SystemComponent. It is the other main type of InternalBehavior and encapsulates the behavioral specifications, which are not of type ExecutionControl. The events and actions in a behavioral specification of this type may not be directly related to the interfaces of the owning ServiceComponent.

AccessPointCommunication - is a generalization and the other main type of ServiceBehavior. It corresponds to notion of externally observable behavior. Behavioral specifications of type AccessPointCommunication specify the actual implementations and use of the interfaces of the owning ServiceComponent. Therefore, the behavioral features of the ServiceComponent, which are related to its interfaces, are referred only in behavioral specifications of this type. AccessPointCommunication extends UML2 *StateMachine* metaclass. As an abstract concept it can be instantiated only through its generalizations.

SAPCommunication - is a state machine type for behavioral specifications describing communication, which is related to provided or used services. SAPCommunication is part of the externally observable behavior of ServiceComponent and SystemComponent. As an abstract concept cannot be instantiated as such, but through its generalizations PSAPCommunication and USAPCommunication.

PeerCommunication - is a state machine type for behavioral specifications describing communication between peer entities, which are part of distributed implementation of a service. PeerCommunication allows keeping the distribution, which is part of the implementation, invisible to the users. In distribution the behavior of a ServiceComponent visible to its users should be preserved as specified initially, i.e., before the distribution. PeerCommunication is part of the externally observable behavior, since this communication takes place on *PeerAPs* of a SystemComponent, which instantiate external logical interfaces. For a ServiceComponent it can be considered as a part of the internal behavior, since it is part of the service implementation in a given network architecture.

PSAPCommunication - type of behavioral specifications, or state machines, specify communication with the external users of the provided services. Events and actions in a state machine of this type are related to the interfaces on the corresponding PSAPs. These interfaces have *implement* relationship to the owning ServiceComponent. PSAPCommunication state machine has peer-state machines in the external entities using these services. These peer-state machines are of type *USAPCommunication*.

USAPCommunication - type of behavioral specifications, or state machines, specify communication with the external providers of required services. Events and actions in a state machine of this type are related to the interfaces on the corresponding USAPs. These interfaces have *use* relationship to the owning ServiceComponent. USAPCommunication state machine has peer-state machines in the external entities providing the requested services. These peer-state machines are of type PSAPCommunication.

3.4 Extensions

LyraUseCase - is a functional model element describing the system functionality. It extends UML2 class *UseCase (from UseCases)* which extends *BehavioredClassifier (from BasicBehaviors)* with the capability to own use cases. Realization relationship between the supplier (LyraUseCase) and the client (SystemComponent) defines how the model elements at different levels of abstraction are related to each other. SystemComponent realizes a LyraUseCase.

LyraActor - is an external entity that interacts with the system. It extends UML2 class *Actor (from UseCases)*, which is extending *Classifier (from UseCases)* with the capability to own use cases.

The summary of the Lyra profile is given in Fig. 2.

4 Ensuring consistency using B Method

4.1 The B Method – background

The B Method [3] (further referred to as B) is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [17]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [18], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

The development methodology adopted by B is based on stepwise refinement [19]. 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, called *refinements*. A formal specification is a mathematical model of the required behaviour of a (part of) system.

In B a specification is represented by a set of modules, called Abstract Machines. An abstract machine encapsulates state and operations of the specification and as a concept is similar to a module or a package.

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. All types in B are represented by non-empty sets.

The operations of the machine are defined in the OPERATIONS clause. B statements that we are using to describe a state change in operations have the following syntax:

```
S  == x := e | IF cond THEN S1 ELSE S2 END | S1 ; S2 |
      x :: T |S1 || S2 | ANY z WHERE cond THEN S END | ...
```

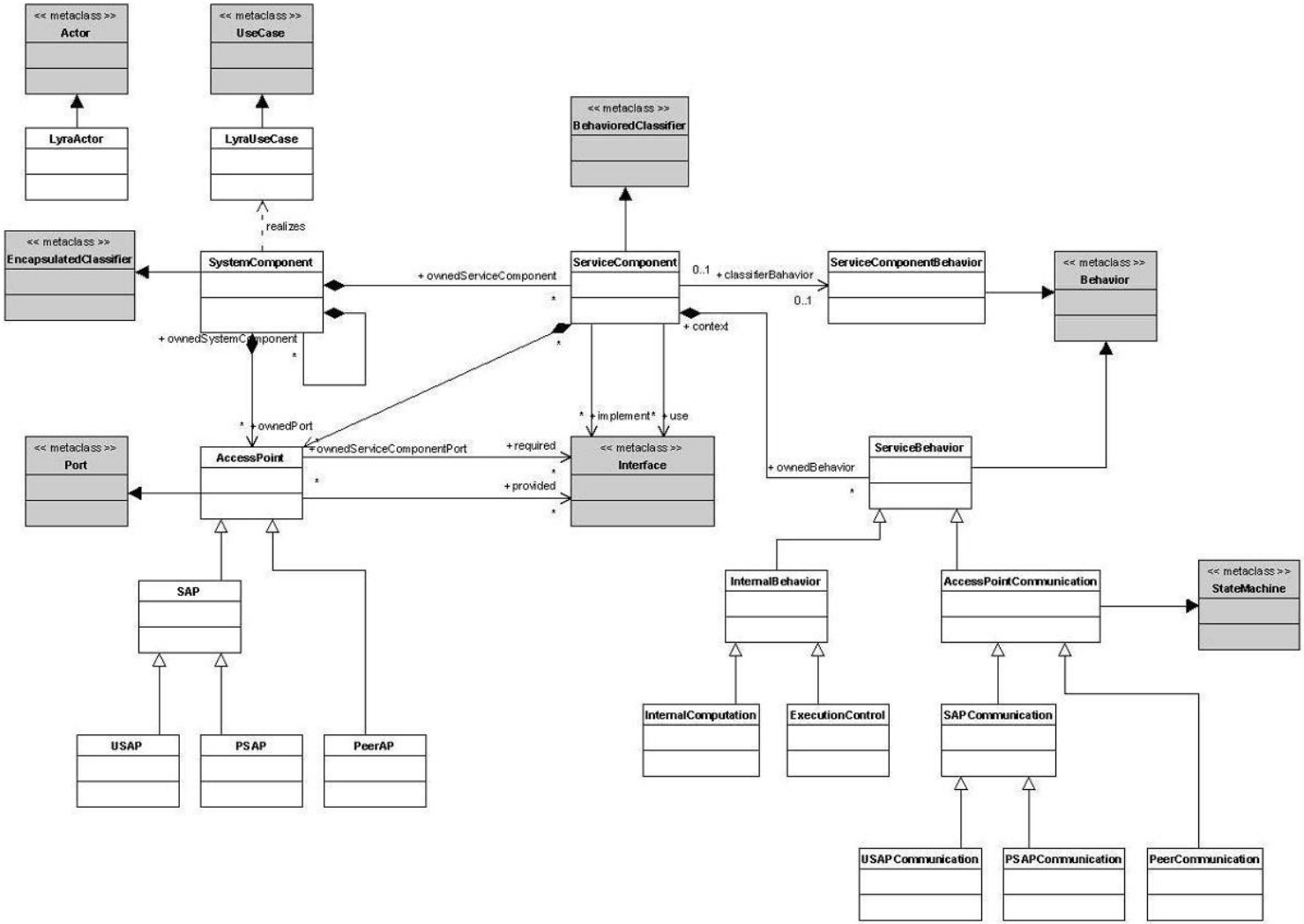

Fig. 2. Summary of the Lyra Profile

The first three constructs – assignment, conditional statement and sequential composition (used only in refinements) have the standard meaning. The remaining constructs allow us to model nondeterministic or parallel behaviour in a specification. Usually they are not implementable so they have to be refined (replaced) with executable constructs at some point of program development. The detailed description of the B statements can be found elsewhere [3].

Often the operations in B are described as the preconditioned operation PRE cond THEN body END. Here cond is a state predicate, and body is a B statement. If cond is satisfied, the behaviour of the operation corresponds to the execution of its body. However, when an attempt to execute it from a state where cond is false is undertaken the operation leads to a crash (i.e., unpredictable or even non-terminating behaviour) of the system.

The B method provides us with mechanisms for structuring the system architecture by modularization. A module is described as a machine. The modules can be composed by means of several mechanisms providing different forms of encapsulation. For instance, if the machine C INCLUDES the machine D then all variables and operations of D are visible in C. However, to guarantee internal consistency (and hence independent verification and reuse) of D, the machine C can change the variables of D only via the operations of D. In addition, the invariant properties of D are included into the invariant of C.

Next we present our approach to ensuring model consistency via specifying modeling concepts and consistency rules in B.

4.2 Ensuring intra-consistency of Lyra models in B

To describe our approach to ensuring consistency between models by formalizing them in B we first focus on addressing intra-consistency rules. As described above, at each development stage of Lyra a fixed set of UML models is produced. For example, the models produced in Service Specification phase are: Domain Model (presented using UML2 use case diagram), Communication Context models (UML2 class diagrams), PSAPCommunication model (UML2 state diagrams) and System Configuration model (UML2 composite structure diagram) [4, 5, 6].

To ensure intra-consistency between these models we should verify that the models satisfy model presentation rules and that they are not contradictory with each other. This task is two-fold. On the one hand, for each individual model we should define model-presentation rules, i.e., the constraints expressing how to define each model element and model itself properly. On the other hand, we should define constraints postulating consistency rules between the models and their elements on each development stage. Fig. 4 presents an excerpt from the definition of these rules for Domain Model and Communication Context model at Service Specification phase. Domain Model and Communication Context model present essentially the same information: Domain Model is considered as an informal description in Lyra, which is used to draft the formal specification described in a Communication Context model.

In our approach to formal verification of intra-consistency, each development stage is represented as a corresponding machine of the general form given in Fig. 3.

The variables of the machine define the corresponding models and their elements. In the invariant clause we list model presentation and intra-consistency rules as predicates over the variables of the machine. Each operation specifies the creation of the model with the enforced consistency rules. In Fig. 5 we present an excerpt from a formal specification of intra-consistency rules between Domain Model and Communication Context model at Service Specification phase.

```
MACHINE INTRACONSISTENCY_STAGE
...
VARIABLES
        Names of models and their elements
INVARIANT
        Model presentation rules &
        Consistency rules for elements and models
OPERATIONS
CREATE_MODEL_A =
PRE conditions for creating MODEL_A
THEN create a model by proper instantiation of its elements
     while ensuring model presentation and consistency
END;
CREATE_MODEL_B = ...
...
END
```

Fig. 3. General form of the intra-consistency rules specification

Domain Model and Communication Context are represented as the corresponding variables DomainModel and CommContext of the machine Intraconsistency_SS. DomainModel is composed of Actor, UseCase, System and Association elements which in their turn are also specified as the variables. The elements of CommContext are defined in the similar way. Observe that the variables of the machine form a hierarchical data structure. Such a structure allows us to define the required additional properties of elements, e.g., their attributes.

In the INVARIANT of the machine we define naming conventions and postulate that each model and its elements are strictly identified by their unique identifiers (UNIQUE_ID). Moreover, we also express constraints associated with each modeling element. We translate each constraint described in natural language in Fig. 4 into B.

Elements of ***Domain Model***	Constraints
Actor	{1} The actor has the name.
Use case	{2} The use case should have a name (the name of the service).
Association	{3} Associates the actor with a concrete use case.
System	{4} Gives the name of the system.
Elements of ***Communication Context***	Constraints
Active class	{5} One active class is created for the system which is defined in the domain model. {6} The name of system is the name of the active class. {7} For each use case in the domain model an active class is defined. {8} The name of the class is the same as the name of the corresponding use case. {9} There is only ONE active class defined for each use case.
External class	{10} For each external actor in the domain model one external class is created. {11} The name of the actor in the domain model becomes the name of the external class.

Fig. 4. Examples of intra-consistency rules

```
MACHINE             Intraconsistency_SS

SETS                NAMES

CONSTANTS           UNIQUE_ID, cc

PROPERTIES          UNIQUE_ID <: NAT & cc:NAMES

VARIABLES
        /* Models */
        DomainModel, CommContext, <other diagrams in the model>

        /* Elements of the DomainModel */
        Actor, Actor_Name, UseCase, UseCase_Name, Association, Association_Source, Association_Target, System, System_Name,

        /* Elements of the CommContext */
        ActiveClass, ActiveClass_Name, ExternalClass, ExternalClass_Name, PSAP_Port,PSAP_Port_Name, USAP_Port,USAP_Port_Name,
        Interface_IN,Interface_IN_Name, Interface_OUT, Interface_OUT_Name

        < Elements of the other diagrams in the model>
INVARIANT
        DomainModel <: UNIQUE_ID & CommContext <: UNIQUE_ID & StateDiagram <: UNIQUE_ID &

        /* Elements of the DomainModel */

        Actor : {DomainModel} <-> UNIQUE_ID &
                /* Actor attributes */          Actor_Name : ran(Actor) --> NAMES &
                /* Constraint [1] */            ! xx. (xx : ran(Actor) => Actor_Name(xx) /= cc ) &

        UseCase : {DomainModel} <-> UNIQUE_ID &
                /* UseCase attributes */        UseCase_Name : ran(UseCase) --> NAMES &
                /* Constraint [2] */            ! xx. (xx : ran(UseCase) => UseCase_Name(xx) /= cc ) &

        Association : {DomainModel} <-> UNIQUE_ID &
                /* Association attributes */    Association_Source : ran(Association) --> UNIQUE_ID &
                                                Association_Target : ran(Association) --> UNIQUE_ID &
                /* Constraint [3] */
                ! zz. (zz : ran(Association) => Association_Source(zz) : ran(Actor) & Association_Target(zz) : ran(UseCase) ) &

        System : {DomainModel} <-> UNIQUE_ID &
                /* System attributes */         System_Name : ran(System) --> NAMES &
                /* Constraint [4] */            ! xx. (xx : ran(System) => System_Name(xx) /= cc ) &

        /* Elements of the CommContext */

        ActiveClass : {CommContext} <-> UNIQUE_ID &
                /* ActiveClass attributes */    ActiveClass_Name : ran(ActiveClass) --> NAMES &
                /* Constraint [5,6] */
                ! xx.(xx:ran(System) & ActiveClass/={}=>#yy.(yy:ran(ActiveClass) & ActiveClass_Name(yy)=System_Name(xx))) &
                /* Constraint [7,8] */
                ! xx.(xx:ran(UseCase) & ActiveClass/={}=>#yy.(yy:ran(ActiveClass) & ActiveClass_Name(yy)=UseCase_Name(xx))) &
                /* Constraint [9] */
                        ! (xx,yy1,yy2). (xx : ran(UseCase) & yy1 : ran(ActiveClass) &
                        yy2 : ran(ActiveClass) & UseCase_Name(xx) = ActiveClass_Name(yy1) =>
                        UseCase_Name(xx) /= ActiveClass_Name(yy2) ) &

        ExternalClass : {CommContext} <-> UNIQUE_ID &
                /* ExternalClass attributes */  ExternalClass_Name : ran(ExternalClass) --> NAMES &
                /* Constraint [10, 11] */
                !xx.(xx:ran(Actor) & ExternalClass/={}=>#yy.(yy:ran(ExternalClass) & ExternalClass_Name(yy)=Actor_Name(xx))) &

        < Invariants for the other model elements >

INITIALISATION
        DomainModel :: POW(UNIQUE_ID) || CommContext :: POW(UNIQUE_ID) || StateDiagram :: POW(UNIQUE_ID) ||
        Actor := {} || Actor_Name := {} || UseCase := {} || UseCase_Name := {} ||
        Association := {} || Association_Source := {} || Association_Target := {} ||
        System := {} || System_Name := {} ||
        <Initialization of the other modeling elements>
OPERATIONS
CreateDomainModel =
 PRE
        DomainModel={}
 THEN
        Actor :: {{DomainModel}*UNIQUE_ID} || Actor_Name :: ran(Actor) --> ( NAMES - {cc} ) ||
        UseCase :: {{DomainModel}*UNIQUE_ID} || UseCase_Name :: ran(UseCase) --> ( NAMES - {cc} ) ||
        Association :: {{DomainModel}*UNIQUE_ID} ||
        Association_Source :: ran(Association) --> ran(Actor) ||
        Association_Target :: ran(Association) --> ran(UseCase) ||
        System :: {{DomainModel}*UNIQUE_ID} || System_Name :: ran(System) --> ( NAMES - {cc} )
 END;
CreateCommContext =
 PRE
        DomainModel/={} & CommContext={}
 THEN
        <Iterate for the number of use cases in the DomainModel>
                <Generate some unique identifier NN> ||
                /* create an active class */
                ActiveClass := ActiveClass \/ ({CommContext}|-> NN) ||
                <Assign the name of the use case to the ActiveClass_Name>
        <End of iteration>  ||
        <Create other elements of the CommContext following the rules for the translation given in the list of constraints>
 END;
< Operations for creating other diagrams in the model>
END
```

Fig. 5. Excerpt from specifying consistency rules in the Service Specification phase

For instance, the following conjunct of the invariant:

```
!xx.(xx:ran(Actor)=>Actor_Name(xx)/=cc )
```

expresses a simple presentation rule for Actor element (constrain {1} in Fig. 4). The more complex constraint {3} which formulates the role of an Association element in the DomainModel is expressed as follows:

```
!zz.(zz:ran(Association)=>Association_Source(zz):ran(Actor) &
        Association_Target(zz):ran(UseCase))
```

By ensuring that the source end of an association is an Actor and the target end of the same association is a UseCase, we identify the *external service provider* in Lyra design flow. Similarly, we distinguish the *service user* by ensuring that the source end of an association is a UseCase while the target end is an Actor.

Observe that the following conjunct:

```
!xx.(xx:ran(UseCase)=>#yy.(yy:ran(ActiveClass)&ActiveClass_Name(yy)=UseCase_Name(xx)))
```

defines consistency rule between elements of Domain Model and Communication Context (constraint {7} in Fig. 4).

Moreover, this rule can be restricted in such a way that the number of associated elements is fixed and limited (constraint {9}):

```
!(xx,yy1,yy2).(xx:ran(UseCase) & yy1:ran(ActiveClass) & yy2:ran(ActiveClass) &
UseCase_Name(xx)=ActiveClass_Name(yy1)=>UseCase_Name(xx)/=ActiveClass_Name(yy2))
```

The operations CreateDomainModel and CreateCommContext specify creating the models at the Service Specification phase. They assign values to the corresponding modeling elements and their attributes. Observe that to enforce consistency we permit to call the operation CreateCommContext only after the Domain Model has been created, i.e., DomainModel/={}, as expressed in the precondition of the operation CreateCommContext.

To verify intra-consistency rules we should prove correctness of the defined abstract machine. To achieve this we use an automatic tool support available for the B Method – AtelierB tool [18]. AtelierB generates the required proof obligations and attempts to discards them automatically. In some cases it requires user's assistance for doing this. Upon discarding all proof obligations the verification process completes.

4.3 Achieving inter-consistency of Lyra design flow via refinement in B

Our approach to consistency verification would be incomplete without addressing verification of inter-consistency, i.e., ensuring that models at different development stages are not contradictory. In this paper we propose refinement as a technique for establishing model inter-consistency.

Refinement [19] is a technique to incorporate implementation details into a specification. A general form of refinement is data refinement, which allows us to change the state space of a machine, e.g., to choose new variables (of possibly different data types) to model the specified behaviour. While replacing abstract data structures with the refined ones, we should define the *linking invariant* which explicitly defines the connection between the newly introduced variables and the variables that they replace. It constitutes a part of the invariant of the refined specification.

Observe that at each development stage of Lyra the rules of intra-consistency remain unchanged. Hence at each stage we can specify the intra-consistency rules by

the corresponding instantiation of the machine Intraconsistency_SS. Then the resultant machines – Intraconsistency_SDe and Intraconsistency_SDi would define the rules of intra-consistency at the Service Decomposition and Service Distribution phases correspondingly.

A specific form of data refinement is superposition refinement [19]. Superposition refinement introduces new variables while leaving the exiting data structure unaffected. Observe that the general ideas of superposition refinement and model transformation during the Lyra development process coincide. Indeed, each development stage introduces a new set of models, while the models created at the previous stage remain unchanged. The linking invariant expresses relation between the existing and newly introduced models. Therefore, by defining inter-consistency rules as the linking invariant and establishing refinement, we verify inter-consistency of models from different development stages. This process is graphically represented in Fig. 6.

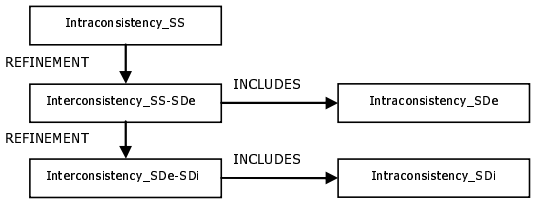

Fig. 6. Handling inter-consistency in Lyra via B refinement

We start from the Intraconsistency_SS machine which specifies intra-consistency rules at the first Lyra phase. We refine this machine by the machine Interconsistency_SS-SDe, which includes the machine Intraconsistency_SDe. The machine Intraconsistency_SDe defines intra-consistency rules for the Service Decomposition phase. On the other hand, the invariant of Interconsistency_SS-SDe defines the linking invariant which contains the inter-consistency rules for models on Service Specification and Service Decomposition phases. The Service Distribution phase is handled in the same way. Due to a lack of space we omit a detailed representation of formal specifications obtained at the refinement process.

In this section we have demonstrated how formal specification and refinement process can assist in formal verification of intra- and inter- model consistency.

5 Conclusion

In this paper we made two major technical contributions. First, we defined the Lyra profile – a profile for architecture-centric development of distributed communicating systems and communication protocols. 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. We defined the basic modeling concepts and established relationships between them. Secondly, we proposed a formal approach to establishing intra- and inter-consistency between Lyra models at different development stages. We demonstrated how to formally express and verify the consistency

rules in the process of specification and refinement in the B Method. The proposed approach establishes a basis for automating the Lyra design flow.

We discussed the related work in Section 2 while presenting the major profiling principles. The major novelty of the proposed approach is that 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 a future work we are planning to further strengthen the proposed approach to automate Lyra-based development of system correct by construction.

Acknowledgments

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

References

1. www.omg.org/technology/documents/modeling_spec_catalog.htm
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. J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
4. S. Leppänen. *The Lyra Method*. Technical report, Tampere Univ. of Tech., Finland, 2005.
5. L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius, Q. Malik. *Formal Model-Driven Development of Communicating Systems*. In Proceeding of International Conference on Formal Engineering Methods, ICFEM'05, Manchester, UK, 2005. (*to appear*)
6. L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius, Q. Malik. *Formal Model-Driven Development of Communicating Systems*. TUCS Technical Report, Number 691, Jun 2005. http://www.tucs.fi/publications/insight.php?id=tLaTrLeLiMa05a&table=techreport
7. J. Rumbaugh, I. Jacobson and G. Booch. *Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
8. UML 2.0 Infrastructure Specification: http://www.omg.org/docs/ptc/03-09-15.pdf
9. OMG, UML Profile for CORBA, formal /02-04-01, Version 1.0, 2002.
10. OMG, UML Profile for Schedulability, Performance, and Time, formal /03-09-01, Version 1.0, 2003.
11. OMG, UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms, formal /04-06-01, Version 1.0, 2004.
12. OMG, UML Profile for Enterprise Distributed Object Computing (EDOC), formal/04-02-01, Version 1.0, 2004.
13. P. Selonen and J. Xu. *Validating UML Models Against Architectural Profiles*. ESEC 2003, 2003, pp. 58-67.
14. P. Selonen. *Model Processing Operations for the Unified Modeling Language*. Doctoral dissertation, Tampere Univ. of Tech., Finland, 2005.
15. C. Riva, P. Selonen, T. Systä, and J. Xu. *UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance*, ICSM'04, 2004.
16. RODIN - Rigorous Open Development Environment for Complex Systems, Project Number: IST 2004-511599, http://rodin-b-sharp.sourceforge.net
17. *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
18. ClearSy, Aix-en-Provence, France. *Atelier B - User Manual*, Version 3.6, 2003.
19. R. J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.