Reusable Case Transformation Rule Specification

Deliverable D3.3, version 1.0, 31.07.2007

IST-2006-033596
ReDSeeDS
Requirements Driven
Software Development System
www.redseeds.eu

Infovide-Matrix S.A., Poland
Warsaw University of Technology, Poland
Hamburger Informatik Technologie Center e.V., Germany
University of Koblenz-Landau, Germany
University of Latvia, Latvia
Vienna University of Technology, Austria
Fraunhofer IESE, Germany
Algoritmu sistemos, UAB, Lithuania
Cybersoft IT Ltd., Turkey
PRO DV Software AG, Germany
Heriot-Watt University, United Kingdom

Reusable Case Transformation Rule Specification

Workpackage	WP3
Task	T3.3
Document number	D3.3
Document type	Deliverable
Title	Reusable Case Transformation Rule Specification
Subtitle	
Author(s)	Audris Kalnins, Elina Kalnina, Edgars Celms, Agris Sostaks, Hannes Schwarz, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Sevan Kavaldjian, Jürgen Falb
Internal Reviewer(s)	John Paul Brogan, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Hermann Kaindl, Daniel Bildhauer, Hannes Schwarz, Kizito Ssamula Mukasa
Internal Acceptance	Project Board
Location	https://svn.redseeds.eu/svn/redseeds/1_DeliverablesSpace/WP3_Reusable-case_specification_language/D3.3/ReDSeeDS_D3.3_Reusable_Case_Transformation_Rule_Specification.pdf
Version	1.0
Status	Final
Distribution	Public

31.07.2007

History of changes

Date	Ver.	Author(s)	Change description
06.06.2007	0.01	Audris Kalnins	Proposition of ToC
03.07.2007	0.02	Audris Kalnins	Changed structure of chapter *Providing models to be transformed*
16.07.2007	0.03	Audris Kalnins	Added initial content for chapter 4
16.07.2007	0.04	Audris Kalnins	Added initial content for chapters 2 and 5
17.07.2007	0.05	Hannes Schwarz	Added content for parts of section 5.2
18.07.2007	0.06	The UL team	Added content for 2.2, 2.3, 5.2.2
18.07.2007	0.07	Hannes Schwarz	Added content for section 5.4
20.07.2007	0.08	The WUT team	Added initial content for sections 3.1, 3.2 and 3.4
25.07.2007	0.09	The UL team	Added MOLA procedures for 4.2
26.07.2007	0.10	The TUW team	Added initial content for section 3.3
27.07.2007	0.11	The WUT team	Added content for sections 3.1, 3.2 and 3.4
27.07.2007	0.12	John Paul Brogan HWU	Conducted document language check and document update.
30.07.2007	0.13	The UL team	Added MOLA procedures for chapter 4.4
31.07.2007	0.14	Audris Kalnins	Added Conclusion
31.07.2007	0.15	TUW team	Added content for section 4.3
31.07.2007	0.16	John Paul Brogan HWU	Conducted document language check and document update.
31.07.2007	0.17	TUW team	Minor changes to section 4.3.

Summary

This deliverable has a relatively narrow focus on using model transformations for building a software case in ReDSeeDS. These software cases are assumed to be built according to the principles of *model-driven* software development. According to these principles, several models are built one after another as artefacts of this software case. This building should be performed according to a well defined *design methodology*, of which we currently are interested only as a set of precisely defined rules, what model elements should be used at which artefacts, what are the naming and structuring rules, what are the desired dependencies between the artefacts and similar issues. It should be specially noted that the starting point of a software case is also a well defined artefact - it is the requirements model in RSL.

Model-driven software development includes as one of the basic principles the use of automatic transformations between design steps whenever this is possible. Thus only the initial version of the next model is obtained, which is then extended manually.

The above mentioned facts on software case development in ReDSeeDS enables the transformations there to have a significantly greater role than in many standard MDSD applications. In many cases the use of automated transformations can be a critical enabler for building a software case according to a strictly defined design methodology in ReDSeeDS.

According to previous deliverables, the SCL includes a special language for defining such transformations - the model transformation language MOLA.

This deliverable illustrates a consistent use of model transformations on an example design methodology, which is based on a 4-layer architecture. For transitions from requirements to architecture model and from architecture to design model initially natural informal transformation rules are provided. Then these rules are implemented in the model transformation language MOLA. For the chosen example methodology, the part of the next model generated by transformations can be especially large. Finally, it is briefly discussed, what is necessary to integrate

the model transformations with other tools and support features for software case development. The full solution of these technology issues goes to Workpackage 5.

It should be noted, that transformations developed in this deliverable are not specific to one software case, but to any case built according to the selected example methodology.

Table of contents

List of figures

Chapter 1

Scope, conventions and guidelines

1.1 Document scope

This document describes transformation rules required for building a software case in ReDSeeDS *software case language* (SCL). The provided rules are based on an example design methodology, which in turn is based on a 4-layer architecture for the target system.

On the one hand, this architecture is very typical for information systems nowadays and therefore can successfully be applied to many software cases. The architecture is described by means of an appropriately chosen subset of UML, some methodology elements prescribe a natural way how these UML elements should be used to define a model, including some naming conventions. On the other hand, this architecture (and the design methodology based on it) is well suited for applying transformations for transition from one step to the next one. In particular, a significantly greater part of the next model can be built by means of automatic transformations, than it is for standard model driven software development. However, a manual refinement of the obtained model is still required at each step.

The deliverable contains a short description of the selected architecture. Then an informal description of the relevant transformation algorithms is provided. A detailed transformation description is given for the following steps:

- transition from requirements in RSL to architecture model in UML
- transition from the architecture model to the detailed design model (both in UML)
- transformation of user interface elements in RSL (directly to detailed design).

Other places where transformations could be applied in software case development are only briefly sketched in this deliverable.

The informal transformation algorithms are implemented in the *model transformation language* MOLA, which is also part of SCL. They support transitions from requirements to architecture model and from the architecture model to detailed design model. The transformation definitions provided in this deliverable have undergone a certain validation, though a complete testing of them will be done in Workpackage 5, when data will be available from the RSL tool. The chosen steps are the most basic ones for demonstrating the ReDSeeDS approach to software case development.

This deliverable discusses also basic principles, which should be used for obtaining models to be transformed from the software case repository and placing the transformation results back to this repository.

The transformations described in this deliverable only depend on the chosen 4-layer architecture and methodology elements related to it. Therefore they are applicable to any software case in ReDSeeDS, which is being developed according to this architecture and methodology elements.

The transformations in this deliverable are typical in the sense that they could serve as examples for transformation development for similar architectures. Especially, the solutions used in MOLA procedures could serve as design patterns for similar situations.

1.2 Conventions

Whenever possible, the language descriptions in this document are based on the corresponding metamodels. The metamodels, in turn, have been developed in accordance with the meta modelling guidelines in deliverable 3.1.

Lowest level package descriptions use the following notation conventions:

- sans-serif font is used for names of classes, attributes and associations, e.g. Requirement
- if a class name is used in description of package other than the one it is included in, it is preceded with package name and a double colon ("::"), e.g. RequirementsSpecifications::Requirement
- ***bold/italics font*** is used for emphasised text, e.g. ***Abstract syntax***

Class colours used on the diagrams indicate membership of the packages. The introduction of colours is intended to enhance readability of diagrams which contain classes from different packages (e.g. a blue colour denotes that classes are from Requirement packages, yellow are from RequirementRepresentation package and green are from DomainElement package). Refer to deliverable 2.4.1 for more information about RSL packages and colours.

MOLA elements are notated in the way used in the initial version of MOLA tool. Metamodels in the MOLA tool use a different colour coding than in SCL definitions. MOLA elements are provided according to the MOLA syntax definition in deliverable 3.2.1.

1.3 Related work and relations to other documents

Model driven development of software has become the de-facto standard for software development in practise, though actually only few books (see e.g.,[KWW03], [SV06]) provide a theoretical background for it. Therefore the systematic transformation based approach in ReDSeeDS to building software cases is quite a pioneering work in this area. The specific transformations to a significant degree depend on the chosen architecture, therefore no close comparison to other sources is possible. A more or less complete description of used transformations for a system development is provided in [KWW03], but a non-standard model transformation language is used there.

Since all languages constituting SCL are defined by means of metamodels, this deliverable significantly relies on deliverable 3.1, where the meta modelling principles for ReDSeeDS were defined. The languages being parts of SCL are described in detail in deliverable 3.2.1. The transformation language MOLA also has a complete description in this deliverable. Therefore the notation and syntax used in this deliverable completely relies on deliverable 3.2.1.

The 4-layer architecture and some transformations related to it were also already used as an example in deliverable 3.2.1. Here they are significantly extended and given a complete description, in order to provide a systematic support for software case development in ReDSeeDS.

1.4 Structure of this document

Chapter 2 is an introduction to the topic of this work. It describes the general role of transformations in ReDSeeDS Software Case development.

Chapter 3 describes the selected 4-layer software architecture model and the defined informal transformation algorithms: from requirements to architecture model, from architecture to detailed design and user interface related transformations (from requirements to design).

Chapter 4 provides the formal transformation definitions in MOLA (from requirements to architecture and from architecture to detailed design).

Chapter 5 discusses the possible solutions for obtaining models to be transformed from the software case repository and placing the transformation results back to this repository. Only the situation when models are stored in JGraLab based repository is discussed currently.

1.5 Usage guidelines

The transformation rule specification document should be used as a book that supplies the software architect with the basic information on transformations applicable for software case development. This document offers one specific 4-layer architecture and methodology elements related to it. If a software case is being developed according to this architecture then the document provides ready-to-use transformations from requirements to architecture model and from architecture to detailed design, as well as transformations for building user interface elements. Complete descriptions of transformations in the MOLA language allow for the modification of some transformation details if required.

If another architecture is selected for the software case, transformations in this document can be used as an example and guidelines for new transformation development for this case.

Users of the this document are expected to know the basics of meta modelling and MOF (Meta Object Facility) specification [Obj06], as well as the syntax and semantics of the MOLA transformation language described in deliverable 3.2.1.

Chapter 2

Introduction

2.1 Role of transformations in ReDSeeDS Software Case development

Due to the requirement of high reusability, software cases in ReDSeeDS should be developed according to well defined design methodologies. Currently there is one such example methodology, based on the 4-layer software architecture model. Most probably, there will be several such methodologies within the ReDSeeDS project. A design methodology prescribes how exactly the vast possibilities of SCL (*software case language*) should be used at various software design steps. In particular, it provides also guidelines how RSL should be used for requirements definition. Just to avoid terminological confusions, by methodology we here understand its part directly related to the development rules and suggested structure of design artefacts, not its broader sense in ReDSeeDS, which involves also human aspects of the design process.

The existence of a well defined methodology provides an important side effect. Since all design steps are built as SCL artefacts in the corresponding subsets of SCL under a strong methodological guidance, there are natural and well defined dependencies between these artefacts in consecutive design steps. These dependencies can be expressed as precise transformation algorithms, how the initial version of the next artefact in the chain can be obtained automatically from the previous one. Certainly, then this initial version of the artefact is manually extended by the software case designer.

The above-mentioned fact ensures a much greater role of automatic transformations in ReDSeeDS software case development, than it is in a standard model driven software development (MDSD). In standard MDSD situations transformations frequently are applied only to some steps - typically from PIM to PSM and from PSM to code skeletons [KWW03]. On the con-

trary, in the ReDSeeDS approach transformations can be successfully applied also to the first step - from precisely defined requirements in RSL to the initial architecture model in UML. Also the next steps from the architecture model to detailed design (which plays the role of PIM in ReDSeeDS) and further to code can be well supported by automatic transformations.

Therefore a special component has been included in SCL for the development of automatic transformations - the transformation language MOLA. The MOLA language is used to define transformations in a well documented and readable way.

Transformations in ReDSeeDS may be of two kinds. A part of them is dependent on the chosen design methodology only and is relevant for all software cases, which have been developed according to this methodology. Other transformations may be software case dependent, which refine the general transformation algorithms for a specific software case (and, consequently, are part of this case). Up to the moment, only transformations of the first kind have been developed.

Automatic transformations have an important role for bolstering case reuse in ReDSeeDS. They guarantee a strict application of the chosen design methodology and, what is especially important, the precise use of automatically generated traceability links. In many situations, the methodology can be strictly adhered to (including all naming conventions) only by means of automatic transformations.

Taking all this into account, the goal of this deliverable is to provide a first set of executable transformations, which would be practically usable for the example design methodology, whose development was started within deliverable 3.2.1 and is continued in this deliverable.

Certainly, the practical value of developed MOLA transformations to a great degree depends on the informal transformation algorithms, inspired by the chosen design methodology. First such algorithms were developed already in the deliverable 3.2.1. Now these algorithms have been refined and cover not only transition from requirements in RSL to architecture in UML, but also transition from architecture model to detailed design. Accordingly, the example based methodology has been extended to fully cover the detailed design step. The set of informal algorithms is quite natural and intuitive, well harmonised with the proposed model structure itself. Therefore, it could be expected that practical application of the developed transformations will be successful.

In order to have practical value, the developed transformations must be well integrated with other software case design tools in ReDSeeDS. Mainly, these are editors for different parts of SCL. While the UML part is expected to be supported by Enterprise Architect, currently there is no clear vision of RSL support in ReDSeeDS. To have at least something tangible, the

prototype version of JGraLab repository based editor is taken as a possible counterpart. Though it will have limited functionality (only textual representations) and it is not ready yet, it has one essential positive aspect. It will directly support RSL according to the tool-ready metamodel of RSL, specified already in deliverable 3.2.1. The exact choice of source and target metamodels is crucial for developing transformation procedures in MOLA. The facilities for transferring models from JGraLab repository and back are also investigated in this deliverable, because this is a crucial precondition for developed transformations to be applied to real models.

2.2 Automatic transformations and manual development

In the classic MDSD approach the automatic transformation based approach is always combined with the manual extension of the generated model by the designer. The same situation is in the ReDSeeDS approach. It is completely universally accepted not to expect a transformation which would be able to generate a software case from its requirements.

Therefore within each software case design methodology it is assumed, that transformations provide only the initial version of the next artefact in the chain. Then this initial version is manually extended within the relevant SCL sublanguage - mainly, UML or code (Java). The initial version provided by transformations must be “user friendly” enough to be easily understood and extended by designers.

The percentage of the automatically generated part of the next artefact with respect to its full contents depends on chosen design methodology and the step within it. Thus, in the provided example design methodology a very significant part of the architecture model (when measured against the number of classes, interfaces and messages in sequence diagrams) is generated automatically by transformations provided in this deliverable. However, a significant manual fine tuning of the generated architecture model is required. For example, in most cases the parameters and return types of generated operations (except those for Business Logic layer services) can only be adequately selected by the architect manually.

For the transition from architecture model to detailed design model in the above-mentioned methodology the percentage of the automatically generated part most probably will be lower, but still the static structure of the model can be generated mostly automatically. Only various manual decisions on additional class operations and more associations should be implemented manually after the initial model generation.

One more aspect which could enhance the automatically generated part is the possibility for the designer to annotate the current model. These annotations have no direct semantic meaning for the current model, but are specially oriented towards guiding the transformations the desired way. One such example could be the grouping of RSL notions into packages. However, in standard MDSD practise such annotations frequently are based on simple extensions of the modelling language - UML in this case. UML has the simple stereotype concept (having the name only) and the tagged value concept for this. Strictly speaking, these are not UML 2.0 features (UML 2.0 has a much more advanced and complicated stereotype concept), but most of the UML tools, including Enterprise Architect, support these simple ad-hoc concepts. Then they can be used in a simple way to guide the transformations. Most probably, such a simple extension facility should be added also to RSL.

For automatic generation and manual extension to coexist in the best way, it is necessary to provide also the update mode for transformations, which update the target artefact (model), when the source one is modified. But these update transformations must not overwrite the manual extensions to the target model (unless they are made explicitly obsolete by the source modification, e.g., a deletion of a class or use case). Therefore these update transformations are more complicated than the initial generation ones, more semantic issues must be taken into account. In this deliverable only the initial generation transformations have been provided, because even the example methodology is not stable enough yet to invest the required effort.

2.3 SCL elements where transformations have the most value

As it was already noted, automatic transformations in the given design methodology have different value and power in transitions from different steps. Currently only the above mentioned example methodology can be analysed. But results could be similar for other methodologies too.

In this methodology the most important transformations are those from requirements in RSL to architecture model in UML, a significant part of the architecture model is generated by transformations. Partially it is due to specially selected structure and naming conventions within the architecture model (which appeared already in deliverable 3.2.1). However, these conventions are quite natural and acceptable in practise. Therefore it is expected, that transformations from requirements to architecture model will be of high value for many design methodologies. This fact is important for ReDSeeDS in general, because software cases will be compared by their requirements, and precisely defined (by transformations) dependencies of architecture from requirements will help to identify required changes.

Transformations from architecture model to detailed design typically can have a lesser ratio of generated part in the target model. This step frequently will be the most creative one. However, for the chosen example methodology the generated part actually is quite large. The main role of transformations could be in guaranteeing precise traceability and naming conventions for the generated parts, which is hardly achievable by other means. It should be noted that user interface part of a system can be directly transformed from requirements to detailed design if the requirements are described in sufficient detail.

The last step is the transition from detailed design to code. In a sense, this step is quite similar to the standard step from the PSM model to code in MDSD. Typically, code skeletons (class signature with attributes and methods) are generated in the relevant language (Java, C#). This feature is supported in most UML tools, including Enterprise Architect.

However, the ReDSeeDS approach can offer more. The well organised sequence diagram structure in the detailed design model (it is especially visible in the example methodology) permits to generate the control structure part of method bodies as well. Currently no standard UML tool offers this (there was a similar attempt in earlier versions of Borland Together).

The existence of easily modifiable transformations in MOLA permits to adapt to various code styles, which is the main issue in standard tools. The SCL definition in ReDSeeDS includes also Java metamodel, which makes the application of MOLA possible for this task. The remaining step is to obtain the Java code from its model (abstract syntax), for which there are several solutions based on standard template languages. This step is not analysed in any great detail in this deliverable.

Chapter 3

Informal description of transformation rules

3.1 Transformation-ready SCL architecture example

The Software Case Language (SCL) contains descriptions for requirements, architecture, detailed design and code models. From pure functional requirements based on domain specification we can obtain a code model by use of transformations between intermediate models (architecture, detailed design). SCL specifies rules for transformations of these models (see Figure 3.1). Intermediate models may be analysed and edited in order to include information omitted during transformation (e.g non-functional requirements).

The architectural model, which is a result of transformation from requirements model, is built with some subset of the UML, as transformation rules are limited to such elements of UML as [KSC+07] defines:

- for static's description: Component, Interface, Dependency, Class, Package
- for dynamic's description: Lifeline, Message

A generated model should be corrected and completed by a software architect. There is complete freedom of using UML elements not limited by any rules, but there must be kept a consistency with the requirements specification. Having the architectural model, transformation rules to detailed design model can be applied. The source of these transformations should be

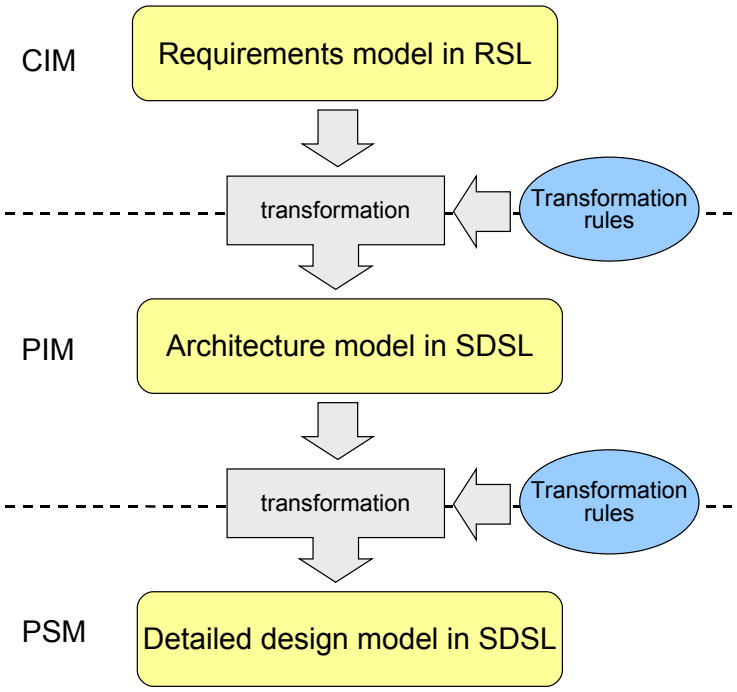

Figure 3.1: Transformations between models defined on different levels of abstraction

an architectural model encapsulating crucial points of the developed software system. It can either be a pure result of a transformation from the requirements model, a result of transformations from the requirements model with some corrections or a complete new architectural model made by a software architect. Transformation rules, based on defined elements of UML used for expressing both static and dynamic descriptions of architecture [KSC+07], while applied on the architecture model, result in a draft of the detailed design model of a developed system. This draft is described, the same as the architectural model, only with some subset of the UML. Elements of the UML used in detailed design model are [KSC+07]:

- for static's description: Class, Interface, Association, Realisation, Dependency
- for dynamic's description: Lifeline, Message, Combined fragments

Generated detailed design models should be analysed and, if needed, edited according to the specifics of the expected results. The skeleton of a detailed design model can be enhanced by the use of any UML elements. From a detailed design model code can be generated by most of the CASE tools.

Having an architectural model based on requirements, there is no need to use a requirements specification to generate a detailed design model. All necessary information is included in the architectural model. The pattern of such an architectural model and an example is presented below, in sections 3.1.1 and 3.1.2.

3.1.1 4-layer architecture model in SCL

As a typical example of an architecture model in SCL, the model of 4-layer architecture will be used. It is the most common multi-layer architecture in today's business software, mostly because of the possibility to distribute logical layers among several machines (servers). The model describes both the static structure - by component and class diagrams, and the system behaviour - by sequence diagrams.

Presentation Tier (UI) - one UI component with one interface is built for the whole application. It mediates between a user and the system, "translating" user-system interaction to calls to the Application Logic tier. Although some logic can be implemented in the UI (like some common sense validation of data entered by user), it is recommended to avoid realising functional requirements in this layer.

The Application Logic layer is responsible for the realisation of rules described in functional requirements of the system. No business logic should be implemented at this tier, only the logic that is needed to control the flow of functional requirements (e.g. Use Cases). The layer structure is designed using component and class diagrams. The application logic layer consists of components, which correspond to logically related groups of use cases. Each use case corresponds to an interface of the corresponding component. Interface contents (operations) are described by means of class diagrams. A components' creation is based on grouping logically related functional requirements. This layer uses Data Transfer Objects (DTOs) for data exchange with the Business Logic layer.

The Business Logic layer is a tier where all business rules of the system are implemented. The above layer, Application Logic, calls interfaces of this tier to retrieve and process data needed to facilitate the flow of user-system interaction. The Business Logic layer calls the Data Access layer for basic objects operations used in performing high-level business data manipulation (for instance validation of sets of data against some aggregated data). The Business logic layer consists of components, which correspond to related groups of domain concepts - notions. For each notion's corresponding interface of this component is created if there are any business methods resulting from functional requirements concerning this notion. This interface is treated as a service providing business methods.

The Data Access layer is a direct access to data source (like database or flat file) of the system. Please note that there can be more than one data source for a given system. In this tier interfaces exposing basic CRUD (Create/Read/Update/Delete) operations on data access objects (DAOs) are implemented for every notion.

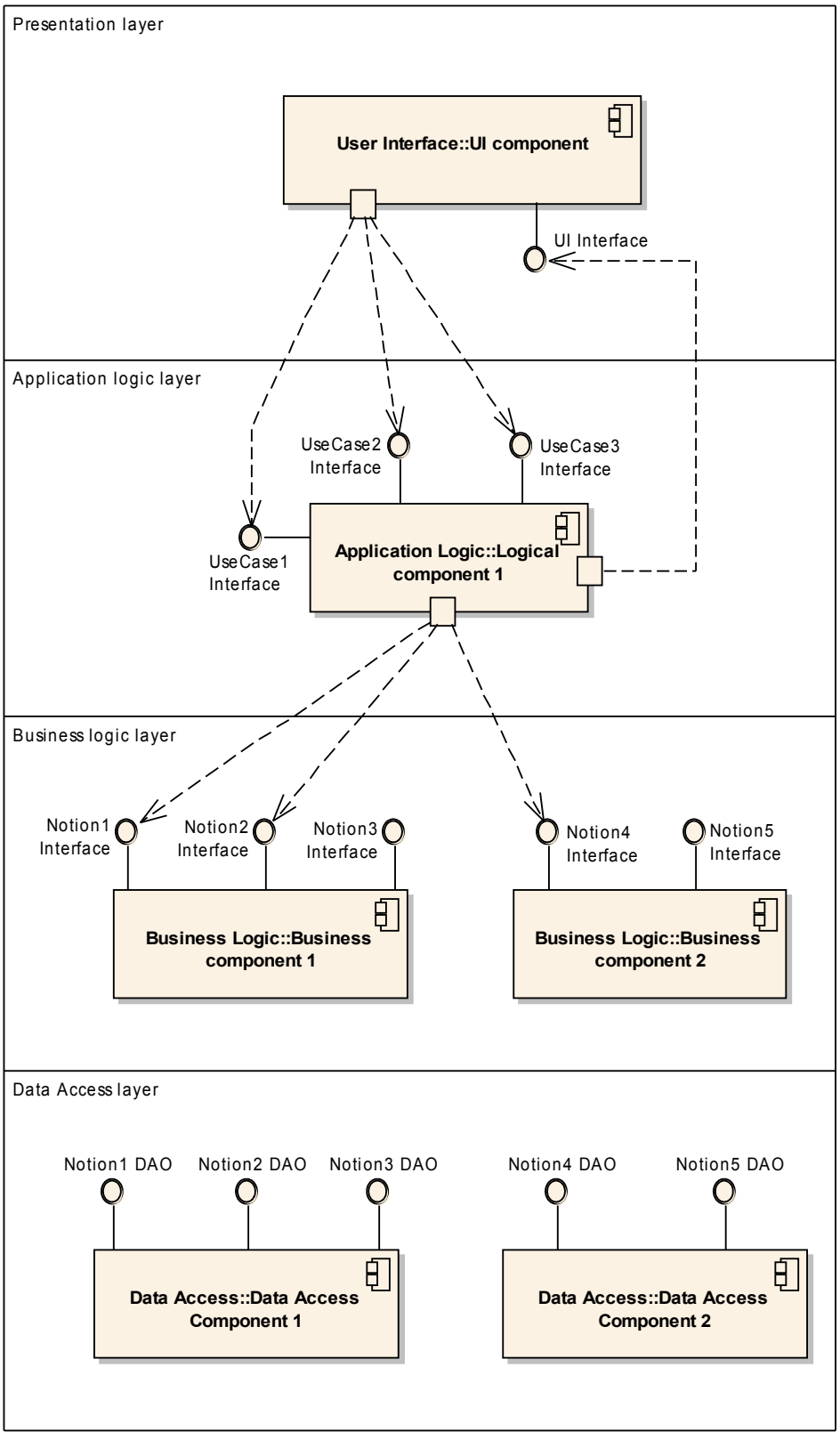

Figure 3.2: Example of 4-layer architecture generated from requirements model

3.1.2 Example of 4-layer architecture in SCL

A part of an architectural model based on the Fitness Club System requirements specification is presented in this section. It is the result of transformations whose the rules are widely described in the sections below. The rules have been applied using an automatic technique, but without using any scripting/transformation tool (by manual editing of the architectural model).

The source for the presented transformation was the RSL Elaborate Example, which is a complete requirements specification written in the RSL for an imaginary Fitness Club software concept. The specification contains 36 requirements – high-, low-level and non-functional, among these are 12 Use Cases, which have several scenarios. The vocabulary part consists of 45 notions with 122 domain statements.

The output of the example transformation is divided into the 4-layer architecture as described in the example from 3.1.1. The resulting model contains 16 components with 62 interfaces having 213 methods and using 40 data transfer objects (DTOs). This gives satisfactory results from the perspective of method-per-interface and interface-per-component ratios. Please note that most of the components and interfaces exist in the Data Access layer, which also has the most complex components (in terms of number of interfaces).

The components with interfaces for all layers are presented in figures 3.4 – 3.7.

The example also contains a few sequence diagrams for selected use cases, that are shown in figures 3.8 – 3.13.

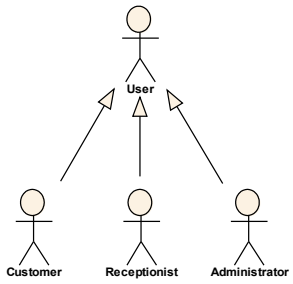

Figure 3.3: 4-layer architecture model example - actors

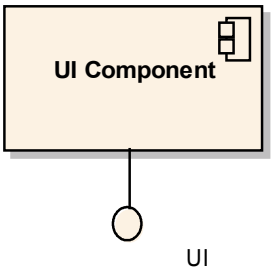

Figure 3.4: 4-layer architecture model example - Presentation layer components with interfaces

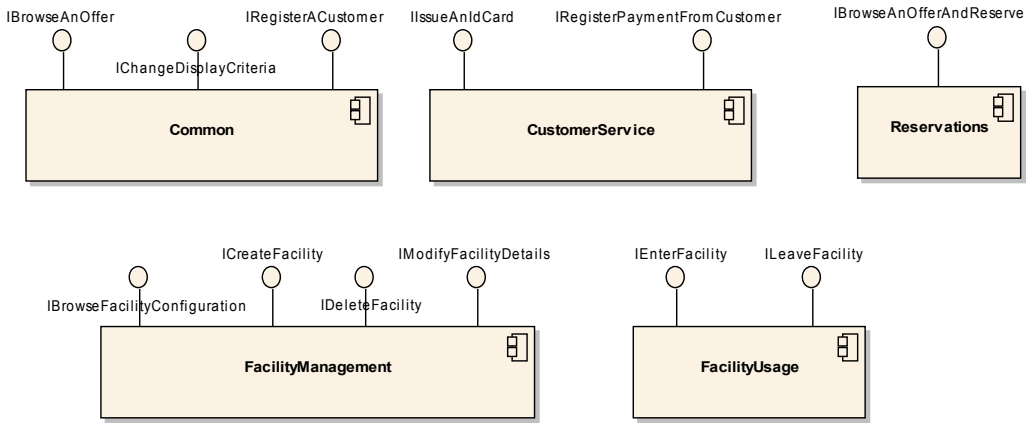

Figure 3.5: 4-layer architecture model example - Application Logic layer components with interfaces

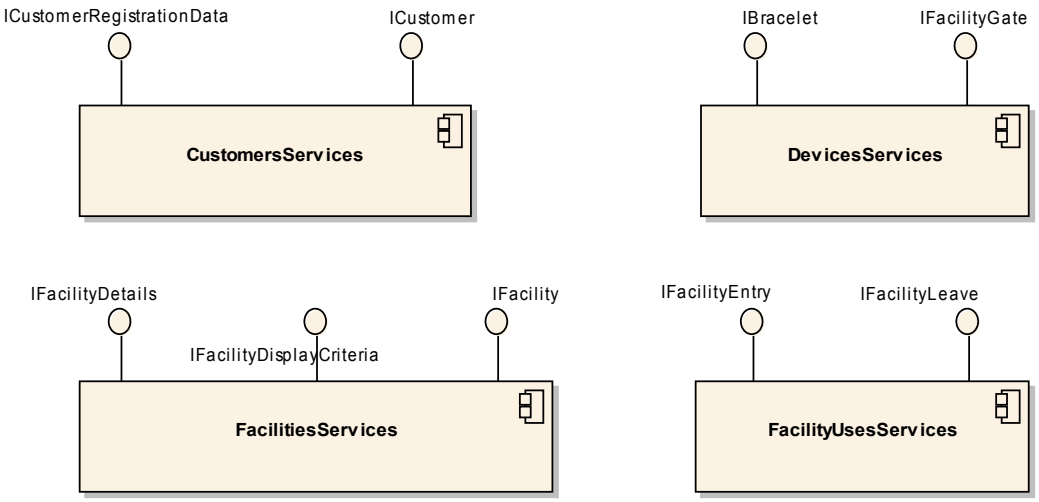

Figure 3.6: 4-layer architecture model example - Business Logic layer components with interfaces

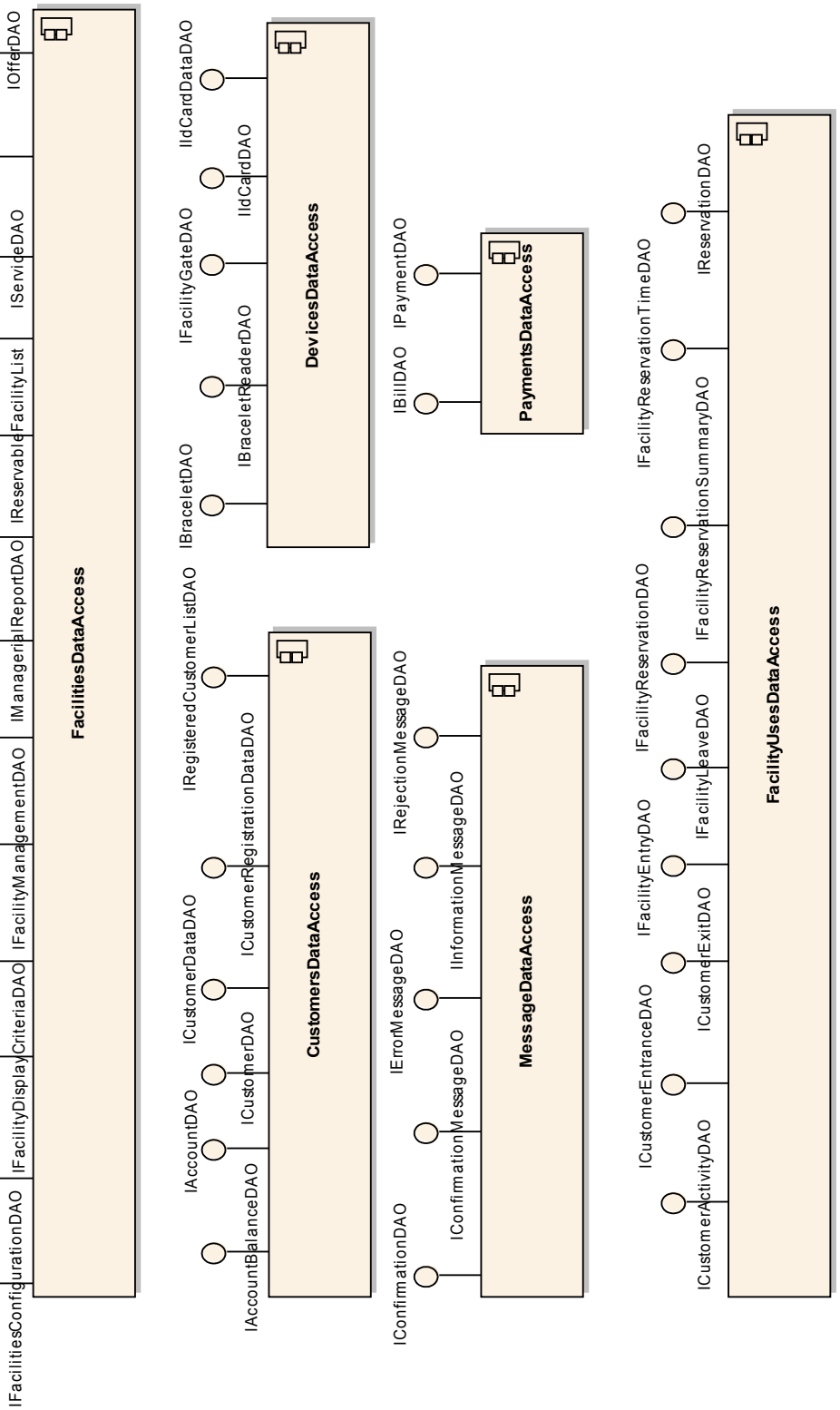

Figure 3.7: 4-layer architecture model example - Data Access layer components with interfaces

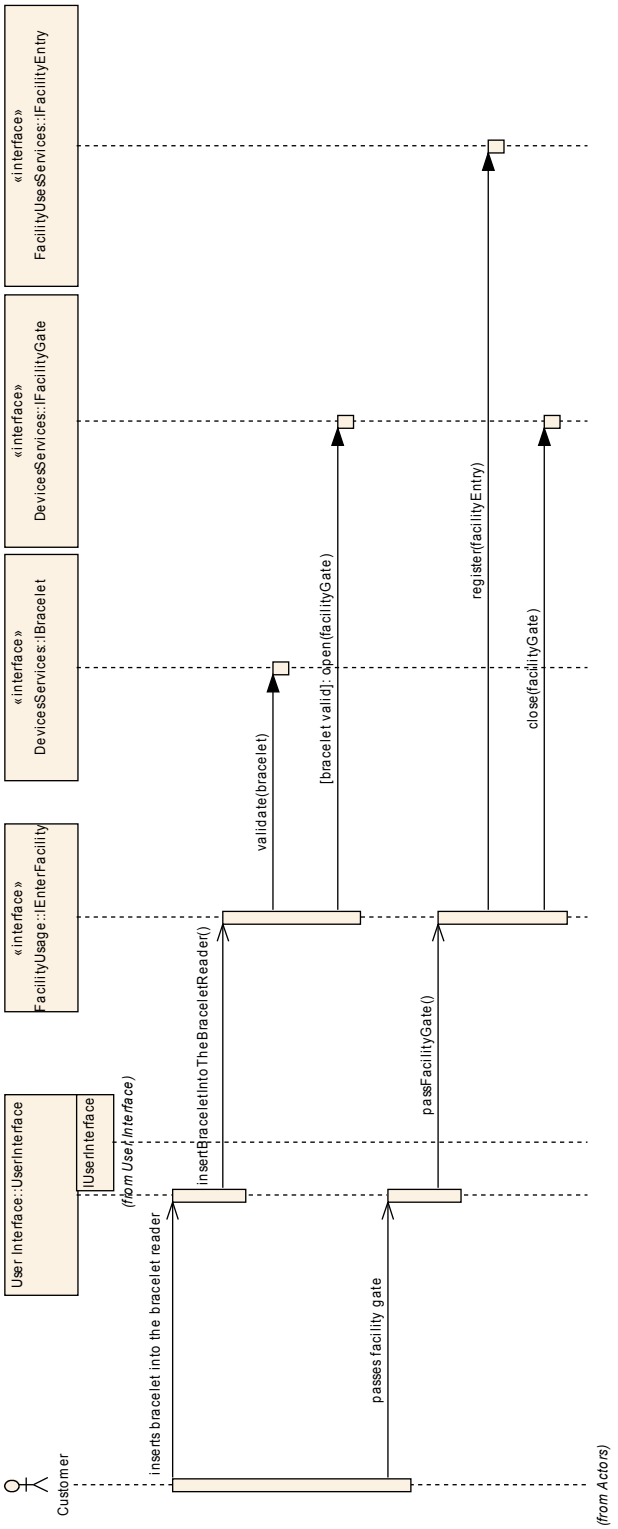

Figure 3.8: 4-layer architecture model example - sequence diagram for basic path scenario of Enter Facility Use Case

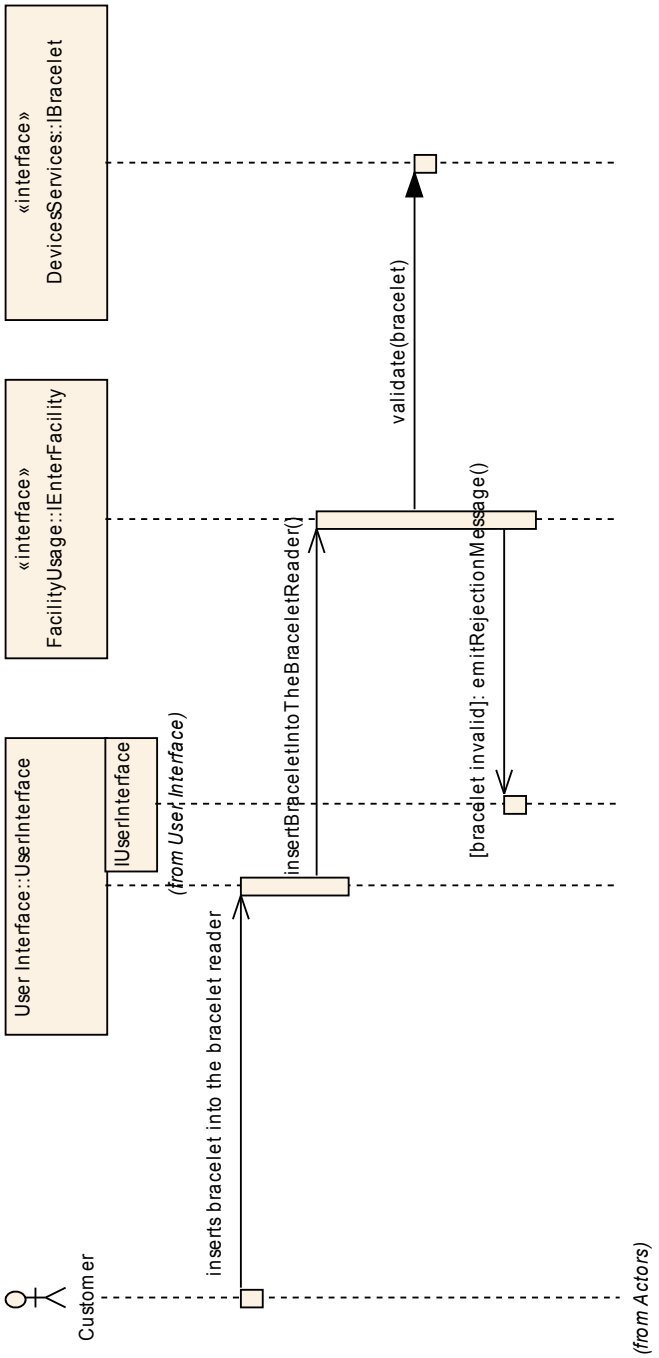

Figure 3.9: 4-layer architecture model example - sequence diagram for 1st alternate path scenario of Enter Facility Use Case

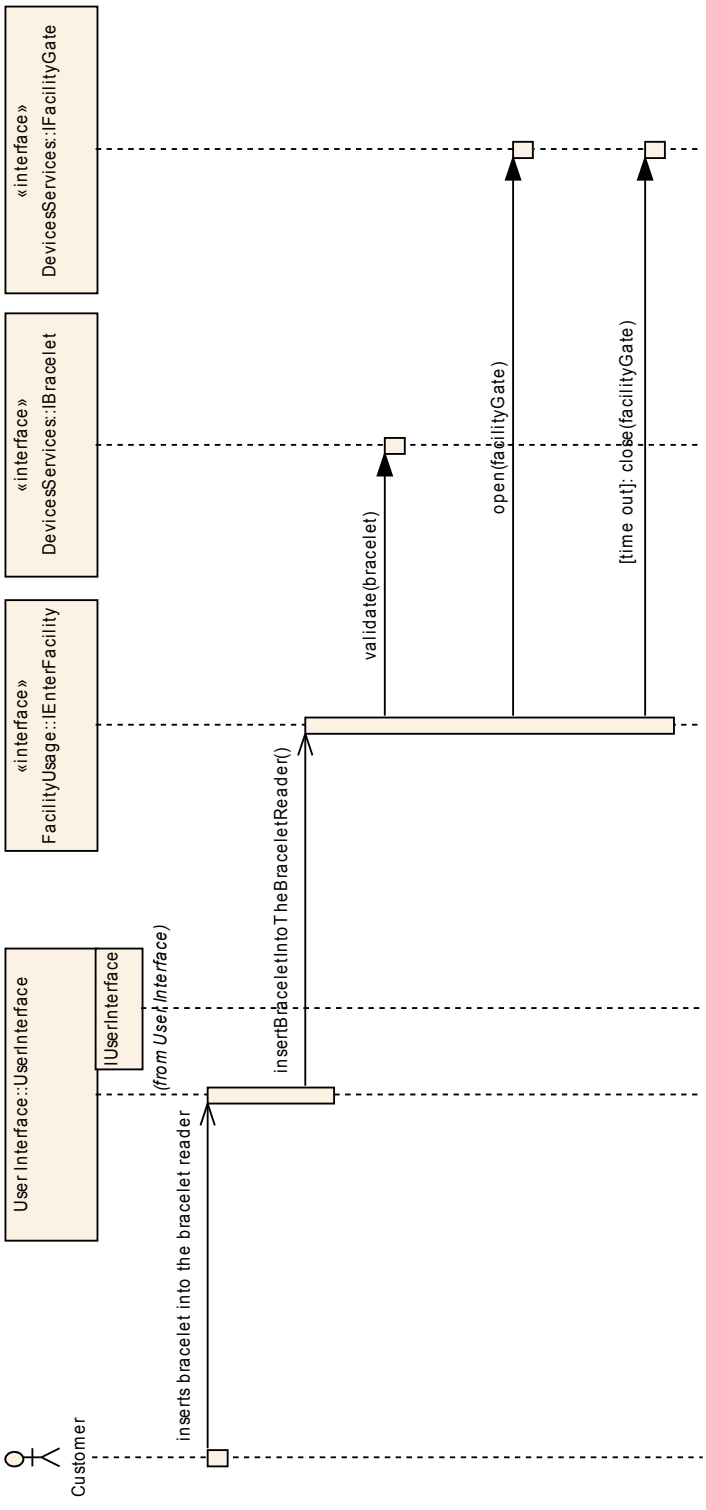

Figure 3.10: 4-layer architecture model example - sequence diagram for 2nd alternate path scenario of Enter Facility Use Case

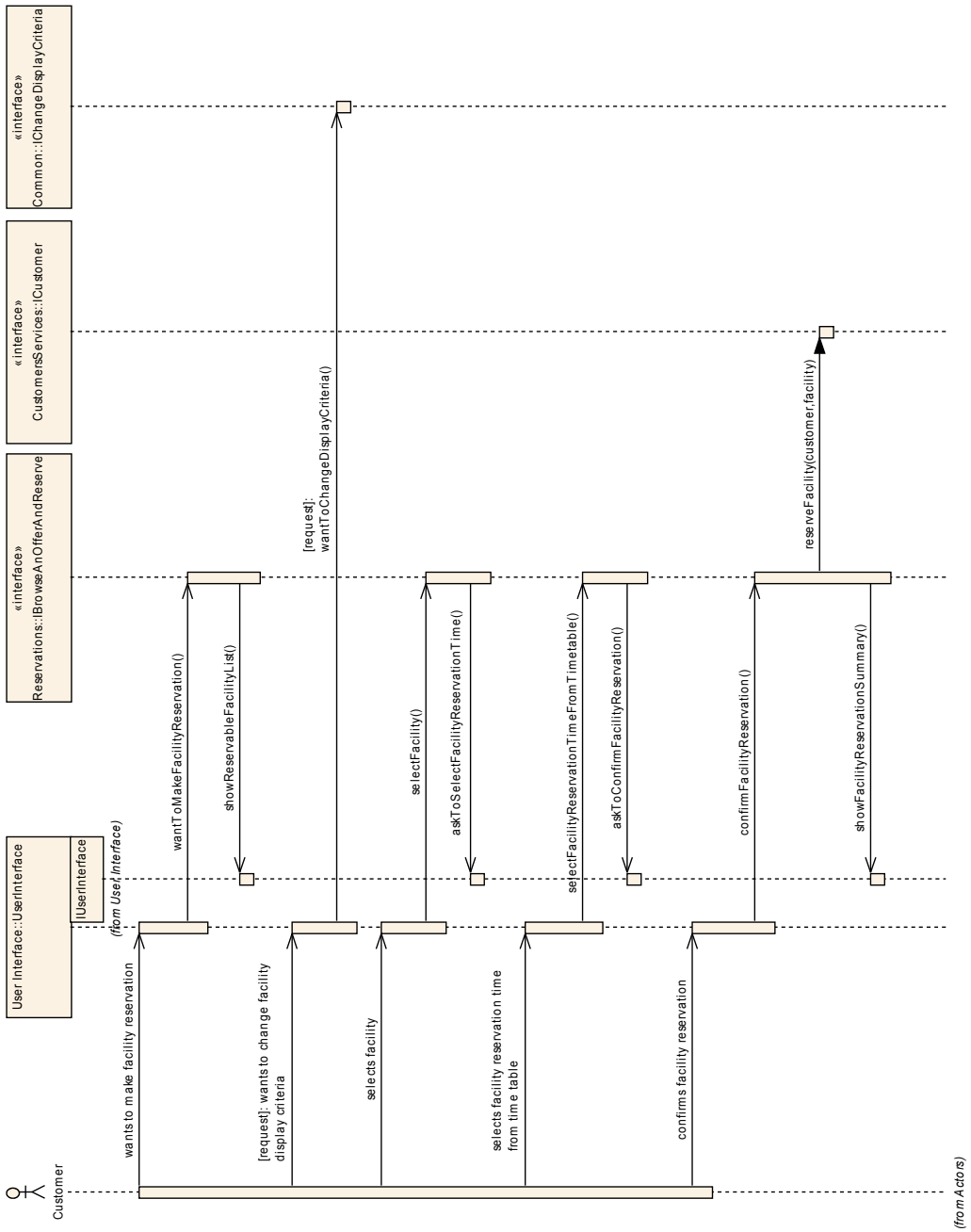

Figure 3.11: 4-layer architecture model example - sequence diagram for basic path scenario of Browse An Offer And Reserve Use Case

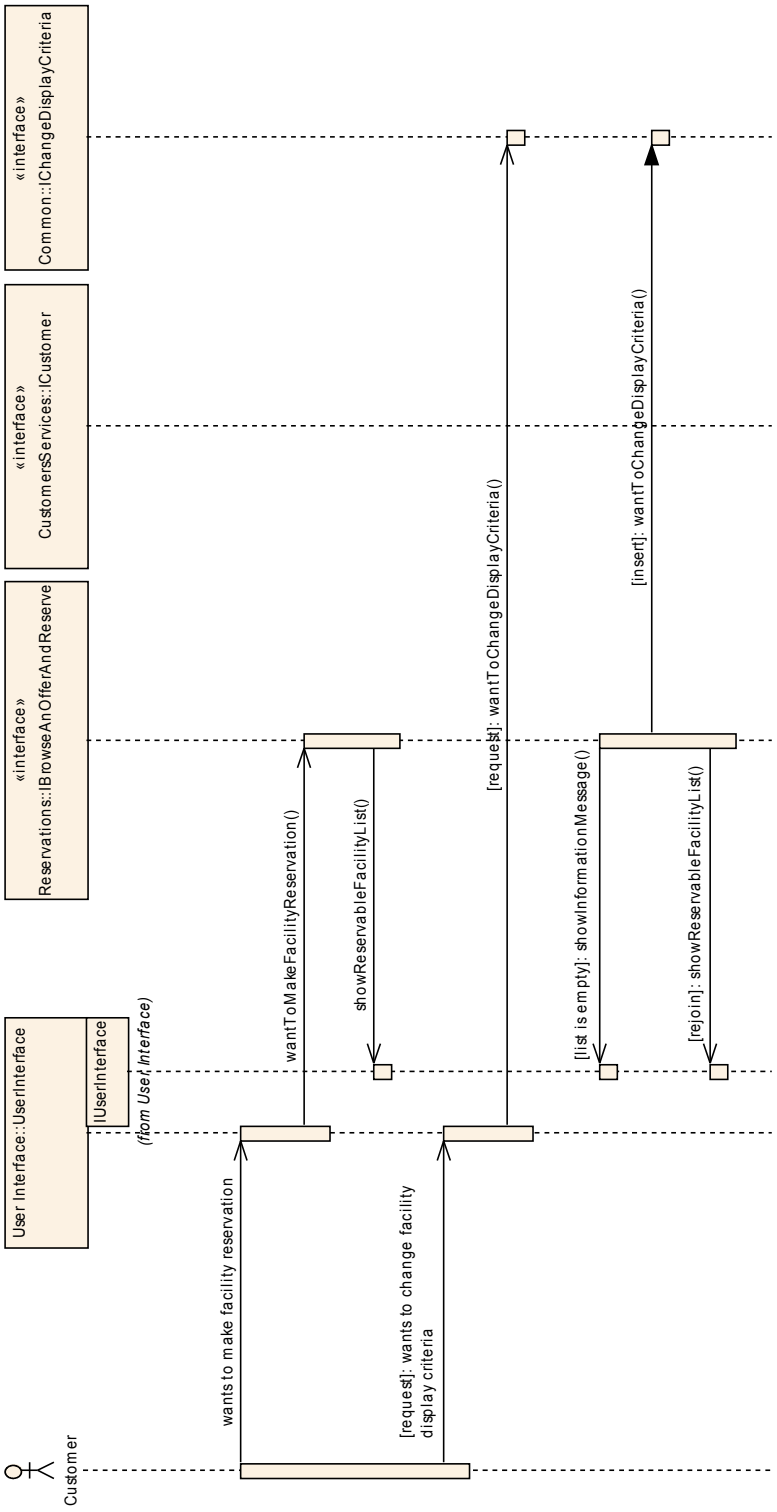

Figure 3.12: 4-layer architecture model example - sequence diagram for 1st alternate path scenario of Browse An Offer And Reserve Use Case

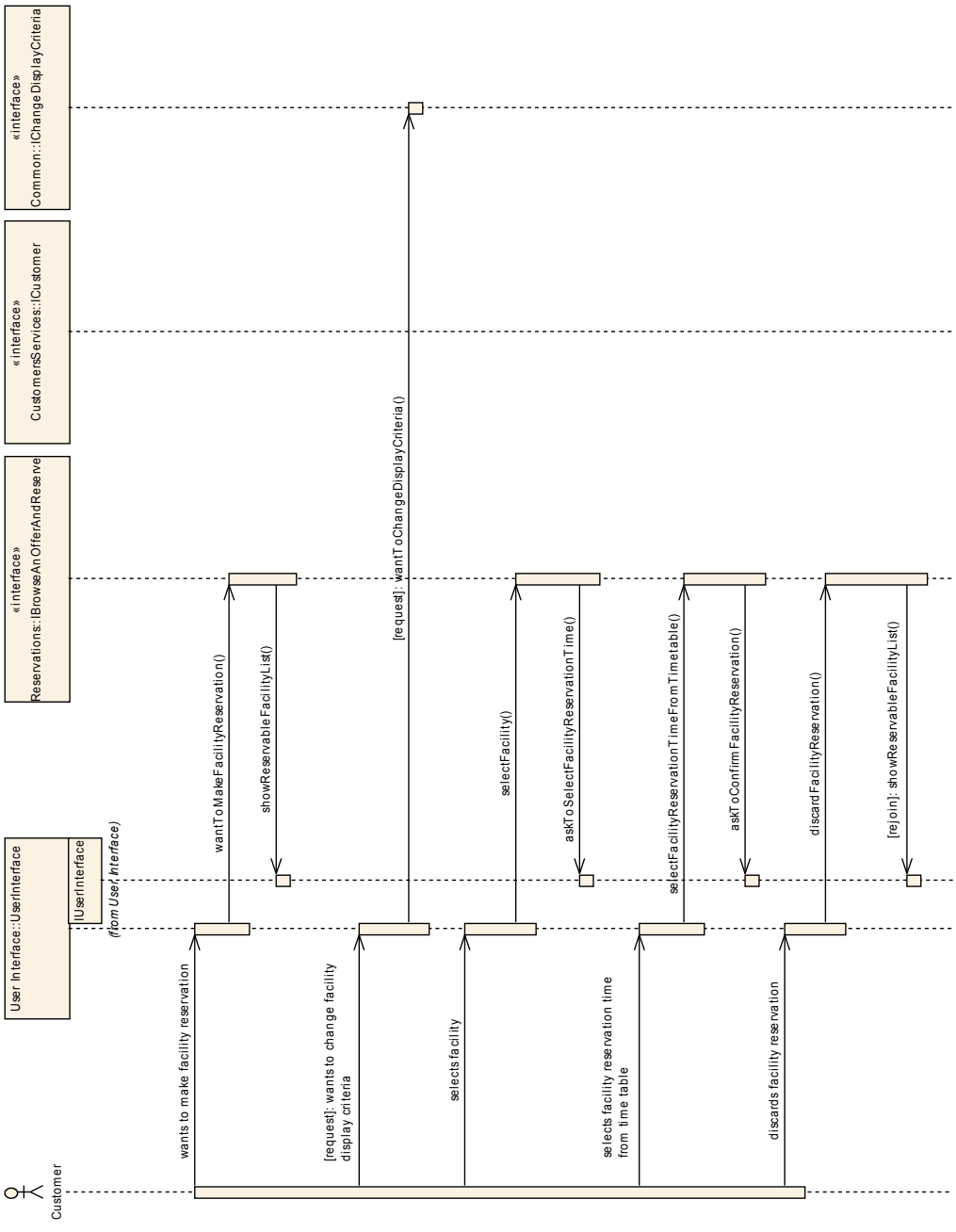

Figure 3.13: 4-layer architecture model example - sequence diagram for 2nd alternate path scenario of Browse An Offer And Reserve Use Case

3.2 Transformations from RSL (requirements model) to architecture model

The purpose of requirements-architecture transformations is generating a draft of an architectural model based on the precise requirements specification. Rules of these transformations should fulfil following criteria:

- the result architecture model should conform to good practises of creating such models (proper granularity of components, good ratio of interfaces per component and methods per interface, etc.)
- changes in source model should have predictable impact on result (target) model
- the rules should be flexible (customisable in part at least)

The precise requirements specification should be the result of collective activities of an analyst and an architect. Some analytical decisions have an important influence on the architectural model and should be made during creation of requirements specification (e.g. grouping vocabulary notions into vocabulary packages and functional requirements into requirements packages).

Having precise, RSL-compliant requirements specification as a source of transformation, a draft of a 4-layer architecture (transformation target, the significance of all tiers of 4-layer architectures is described in section 3.1.1) can be generated by applying following rules[1] (see Fig. 3.14, 3.15):

- every vocabulary package is transformed into one business component and one data access component
- every notion used in an SVO sentence is transformed into a data transfer object (DTO). Notions which occur only in "high-level requirements" and system vision are ignored. Every relation between notions is transformed into association between corresponding DTO classes. Associations between DTO classes reflects direction and multiplicities of notions' relations (if present).
- for every notion used in an SVO sentence, a DAO interface is generated in the corresponding data access component. This interface will provide CRUD operations for any given notion.

[1] This set of rules is refined from [KSC+07]

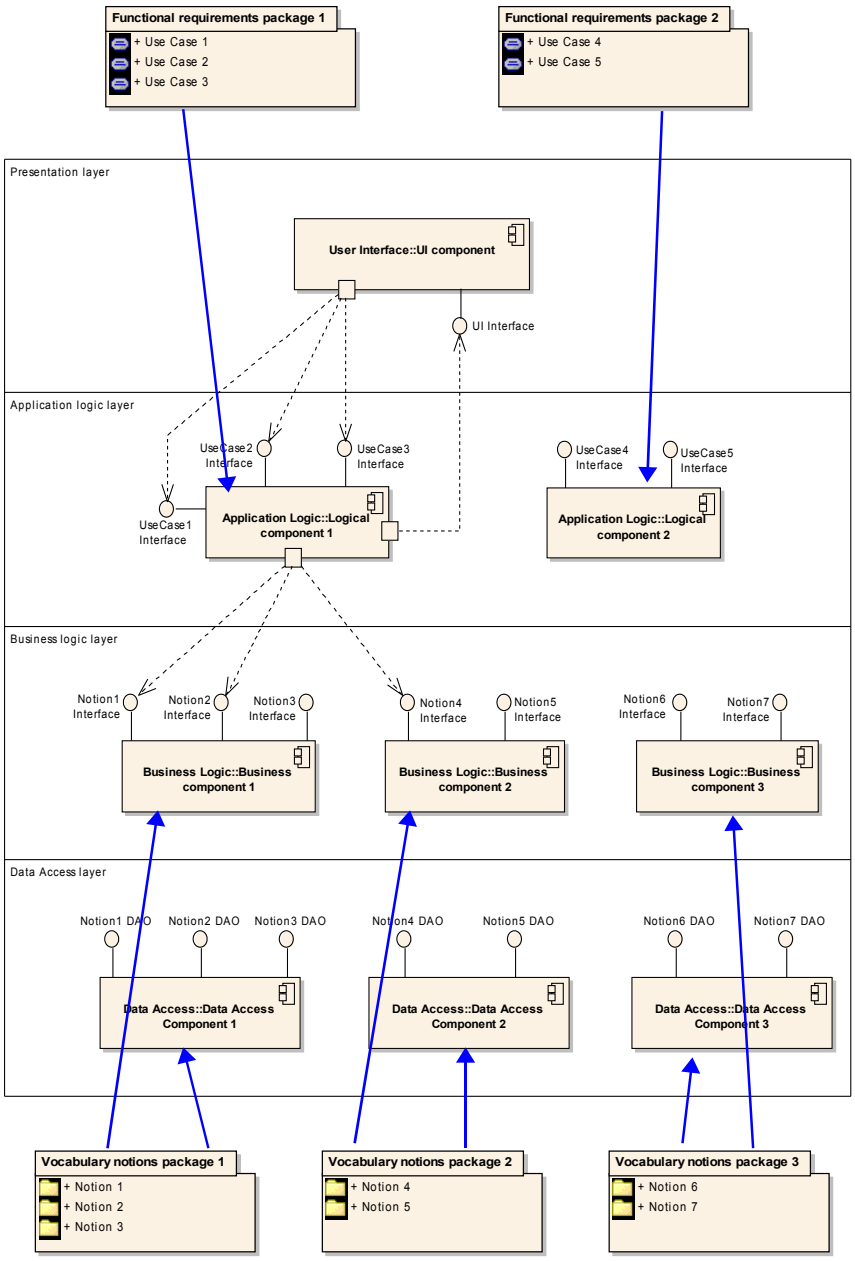

Figure 3.14: 4-layer architecture generation overview

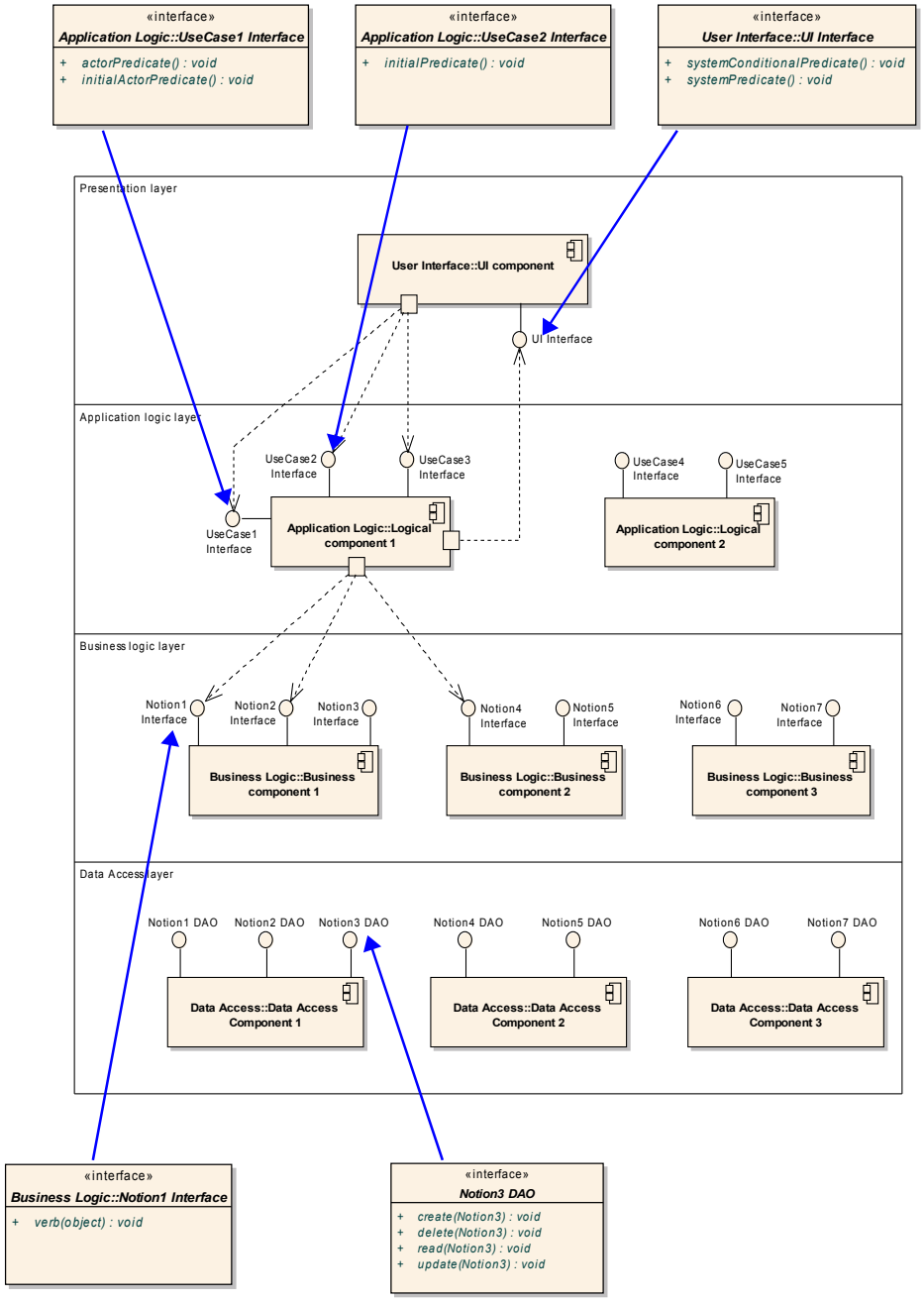

Figure 3.15: 4-layer architecture generation - interfaces

- a notion in a given vocabulary package is transformed into one interface provided by a business component. An interface corresponding to this vocabulary notion is generated only if the method for this notion is called on a business component (in the scenario we have self message with this notion in predicate).
- every functional requirements package is transformed into one application logic component; it is important to keep the number of functional requirements in packages low (for instance: less than 4), to prevent creation of too complex components with too many interfaces (see also next rule)
- every UseCase (functional requirement) is transformed into one interface provided by application logic component
- one UI component with one interface is generated for the whole application; further division of this component can be done by an architect at a later stage, when storyboards are ready and UIElements are placed in the vocabulary
- all actors are transferred with no changes to architectural model

3.2.1 Generating architectural details

To generate further architectural detail, a sequence diagram from a requirements specification needs to be transformed into a sequence diagram for the generated architecture.

In requirements sequence diagram we have several types of messages:

- initial actor predicate - the first message from an actor to the system
- actor's predicate - every message from actor to the system (Every sentence with actor as subject)
- system predicate - every message from the system to the actor (Every sentence with system as subject, followed by sentence with Actor as subject)
- System's "self-message" - every message from the system to itself (Every sentence with system as subject, followed sentence with system as subject)

Based on the above terminology we should apply the following rules of transformation to create a more detailed architectural model (see Fig. 3.16):

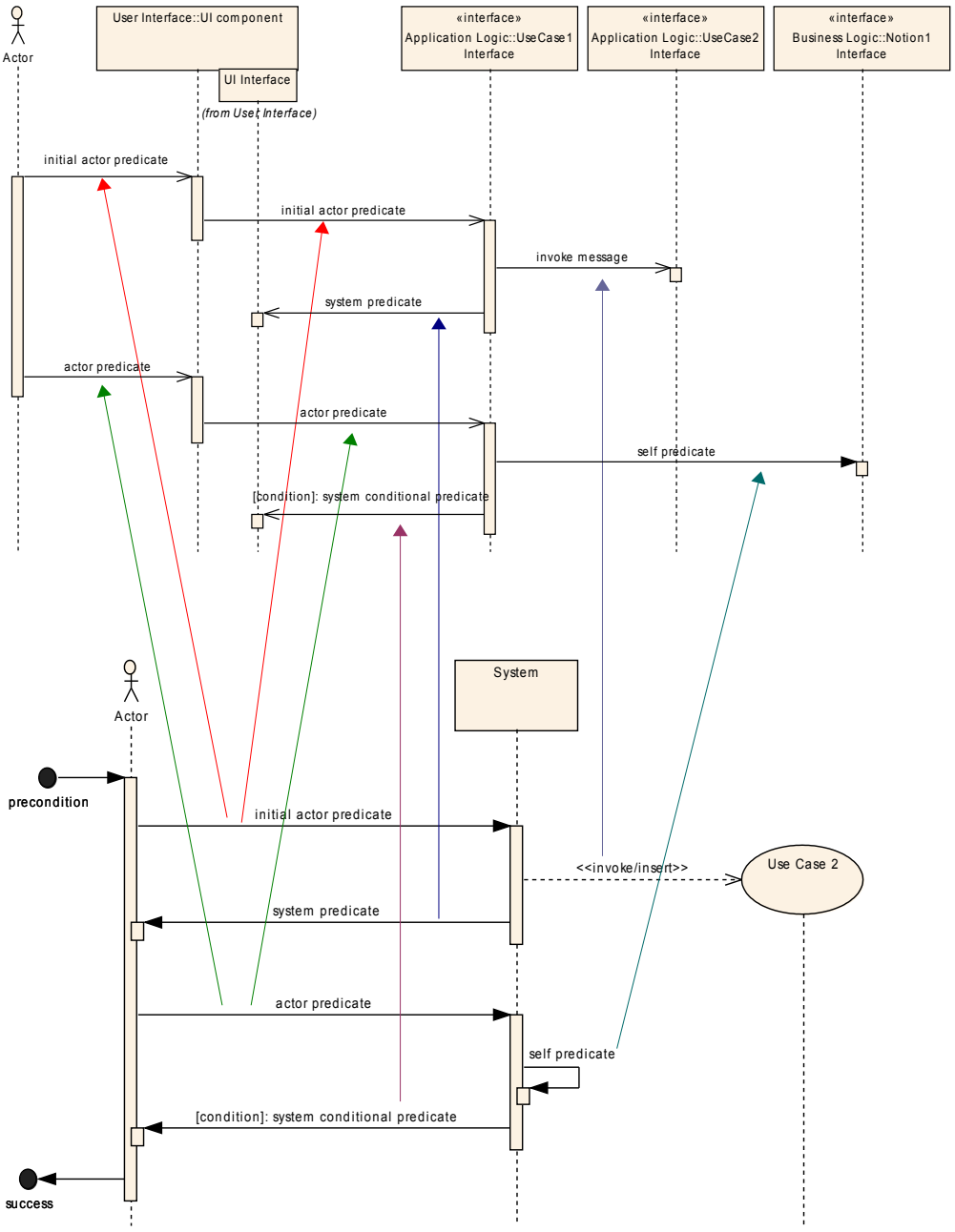

Figure 3.16: 4-layer architecture generation - sequence diagrams

- initial actor predicate (description of an action that initiates a UseCase) is transformed into two calls: one from Actor to presentation layer component and the other one from presentation layer component to application logic component [2]. Also, a dependency between the presentation layer component and the application logic component's interface used in this call should be created.

- each actor predicate is transformed analogously to initial actor's predicate

- every system predicate (description of an action that is a system's response to Actors activity) is transformed into call from application logic layer component to presentation layer component

- every "invoke" construct is transformed into call from "current" application logic layer component (the one for "current" UseCase) to application logic layer component of invoked UseCase

- every System's "self-message" is transformed into call from application logic layer component to business layer component; this business layer component corresponds to a notion which occurs in a predicate in an object. For this call a dependency between "source" application logic layer component and "target" business layer component's interface used in this call should be created in the architectural model.

- each of the above calls is transformed into a method of a related interface

- calls to business logic layer component's interface should correspond to verb phrases of Notion which forms this interface (see Fig. 3.17):

  - if verb phrase in predicate is a simple verb phrase, then verb is used for the method's name, and object part of predicate is a method's parameter (example: [[v: add n: user]] => *add(User)*)
  - if phrase is a complex one, then verb and direct object form method's name and both direct and indirect objects are method's parameters (example: [[v: add n: user p: to n: user list]] => *addUser(User, UserList)*)

- actors in architectural models should be separated from actors from the requirements model (but they should have same names, the architectural model actors are just "copies" of actors used in requirements)

- calls to UI interface are asynchronous by default (but can be changed later by an architect to synchronise if needed)

- no returns are shown on architectural sequence diagrams

[2] Both components are specific for this UseCase - see above

- methods do not return anything - return types should be set at a later stage by an architect

3.2.2 Naming of architectural model elements

Although for names of architectural model elements the RSL requirements model element names could be used, we propose also a set of rules for renaming elements resulting from notions, use cases, verb phrases, etc.:

- all element names should be converted to UpperCamelCase (aka PascalCase), e.g. *user list => UserList*
  Exceptions:
  (1) calls between actor and presentation layer should remain in a form of SVO sentences
  (2) method names should follow CamelCase naming pattern
- in Business Logic layer all component names should have word “Services” added at the end, e.g. *FacilityUses => FacilityUsesServices*
- in all layers all interface names should have letter “I” added at the front, e.g. *FacilityDetails => IFacilityDetails*
- in Data Access layer all component names should be in plural form and have words “DataAccess” added at the end, e.g. *Device => DevicesDataAccess*
- in Data Access layer all interface names should have additionally acronym “DAO” added at the end, e.g. *Account => IAccountDAO*
- all notions (which form data transfer objects) should have names ending with acronym “DTO”, e.g. *RejectionMessage => RejectionMessageDTO*

These rules allow model names to be more conforming to the software naming standards. The rule set should be customisable in the ReDSeeDS tool, as no design pattern or platform specific naming should be enforced on ReDSeeDS framework users.

3.2.3 Manual editing of generated model by an architect

The above transformations aim to generate platform independent model (PIM in the MDA terminology). The platform specific detailed design model can be generated/designed based on

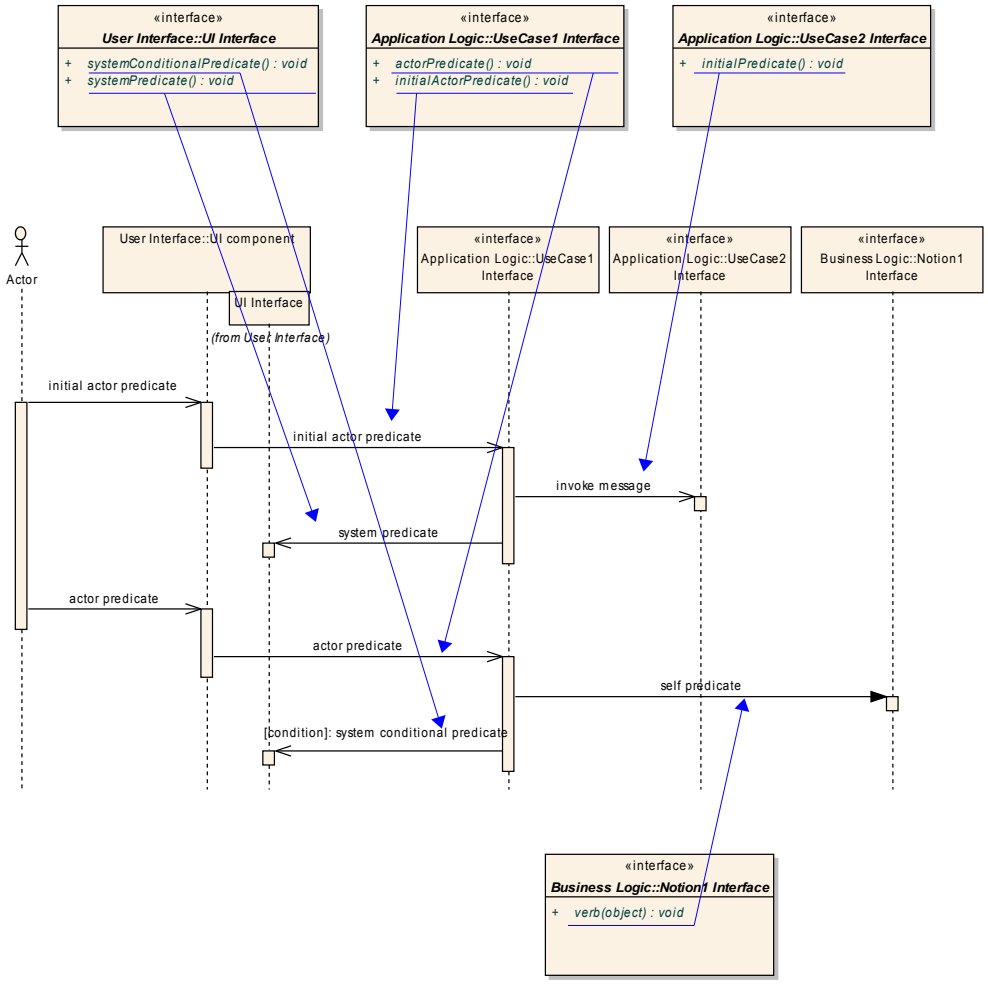

Figure 3.17: 4-layer architecture generation - methods of interfaces

a given architecture (see 3.4), which the architect should carefully examine and improve. For this purpose the architect can use any elements of UML, but should have in mind the fact, that only the SCL-compliant UML subset can be used for later transformation (from architecture to detailed design).

Some of the decisions made by an architect after generation of the architecture model should take into account difficulties with determining which of the notions should have corresponding DAOs in the final architectural model (e.g. collections for existing DAOs should be removed, as typical, auto-generated CRUD methods do not make sense for lists of objects).

In the ReDSeeDS tool a feature should be implemented to mark an architect's changes in the final model, so re-generating the model again from requirements (for example: after requirement changes) would not overwrite the architect's work. The solution could be based on keeping traces between the generated model and the final model. The example of such a link could be IsAllocatedTo ([KSC+07], p. 214). Also, changed elements in a model should be indicated with some kind of "change flag". During re-generation of architecture the architect would be prompted for overwriting their changes. Of course there are still few problems to solve: for example if an architect splits a component into two new components, and after generation a new method appears in the original component, the decision has to be taken: to which of the resulting two components the new method should be added.

Presented transformation rules are appropriate for generating 4-layer architecture. For other architectures (like for the real time systems, service-oriented architectures or client-server architecture) different transformation rules should be proposed. In fact we need a "family" of transformations which probably should be customisable.

While 4-layer architecture seems to be most illustrative for purpose of creating transformation rules, transforming to other architectures following the layer architectural design patterns could be based on rules presented above; e.g. transforming to 3-layer architecture could be very similar with exception of merging UI and application logic layers (calls between UI and application logic become UI layer self-messages).

3.3 Transformations of UI elements in RSL

According to the 4-layer architecture described in section 3.1.1 the presentation layer consists only of one UI component with one interface on the architectural level. Therefore, the transformations described in this section transform the UI elements in RSL to a detailed design model of this generated UI component. This step requires that storyboards are ready and UI elements are placed in the vocabulary, otherwise the detailed design model of the UI component has to be done manually by a designer.

The transformation into a detailed design model consists of two steps. First, an overall structure is generated containing factories, dialogs and interface elements to the application logic. Second, RSL UI elements and storyboards are used to generate the dialog structures.

The following rules describe the transformation into the overall presentation logic structure:

- for the presentation layer one UI factory is created. It is a static class with name “UIFactory”.
  - for the UI interface in the UI component a method returning a realisation of this interface is generated. The method name is composed of the prefix “get” and the component name: “getUI”.
- for the UI interface in the presentation layer a corresponding interface and implementation class are generated together with a realisation relation. The name of the interface gets the prefix “I” and will be “IUI” and the implementation class will be called “UI”.
  - interface methods are the same as in the architectural model. These are methods corresponding to system predicates.
  - for the implementation class also methods corresponding to actor predicates are added, since the UI class also acts as a port to the application logic layer for all other UI classes.
- for the UI factory dependencies to the UI implementation class and to all dialogs and windows are generated.
- for each RSL UI presentation unit contained in a UI scene a dialog class or frame class (depends on whether the UI scene attached SVO scenario sentence contains the modifier “modal” or not) together with its structure and layout are generated. The dialog or frame gets the name of the UI presentation unit in CamelCase notation suffixed with “Dialog”

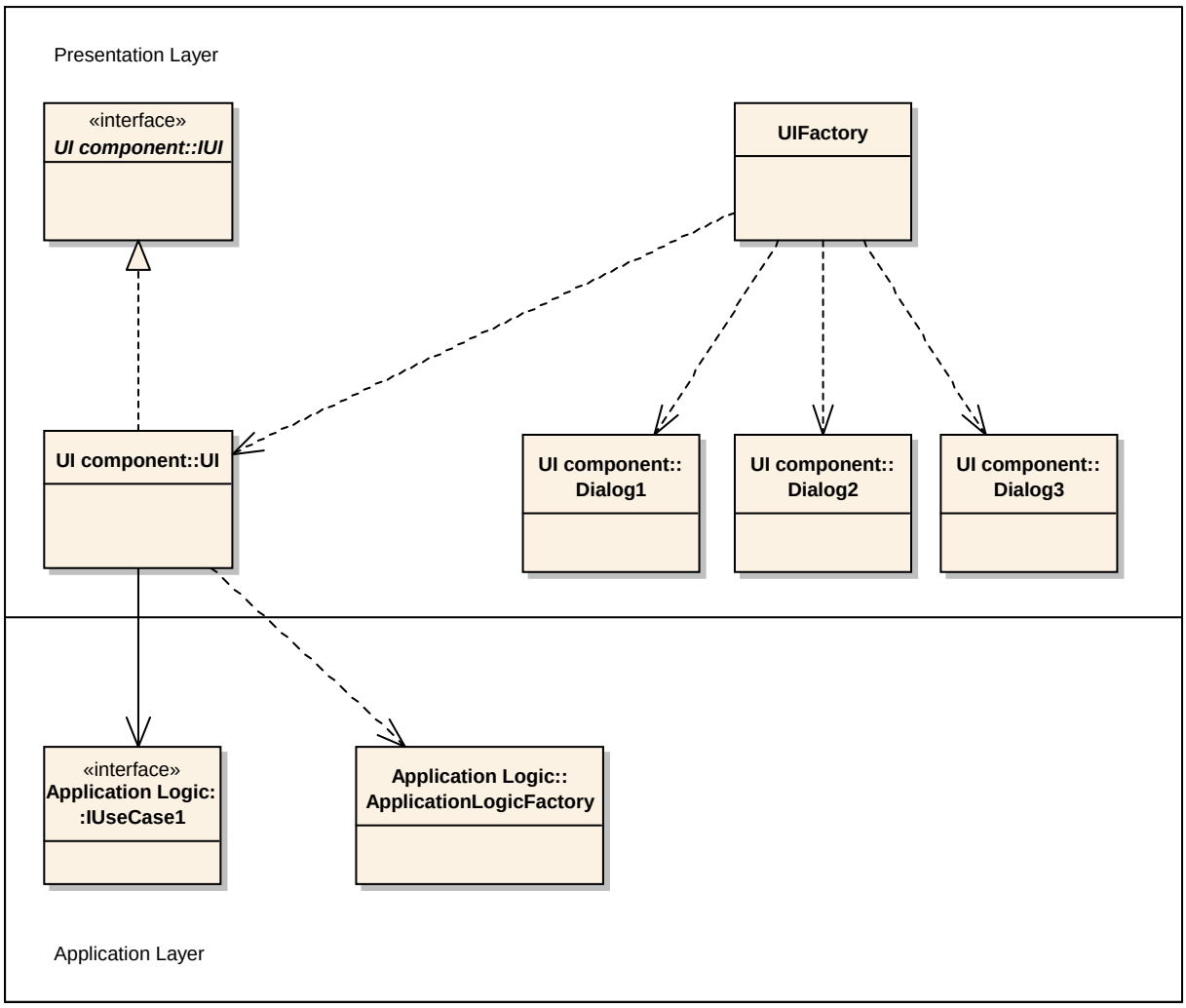

Figure 3.18: Overview of the UI component's detailed design model

or "Frame" (e.g. ReservableFacilityListDialog). Detailed rules for generating the dialog and frame structure are described in section 3.3.1

- for each dialog or frame in the UI component a static method returning a single instance of this dialog or window is generated. This method implements the Singleton-Pattern and its name is composed of the "get" prefix and the dialog's or frame's name (e.g. getReservableFacilityListDialog).

- for each system predicate with the verb "show" the method added to the UI class retrieves the dialog from the UIFactory and displays it to the user. The object part of the predicate shall correspond to the UI presentation unit and thus to the dialog's name.

Figure 3.18 shows the structure of the UI components detailed design model generated according to the rules above and Figure 3.19 illustrates an example for a generated detailed design model.

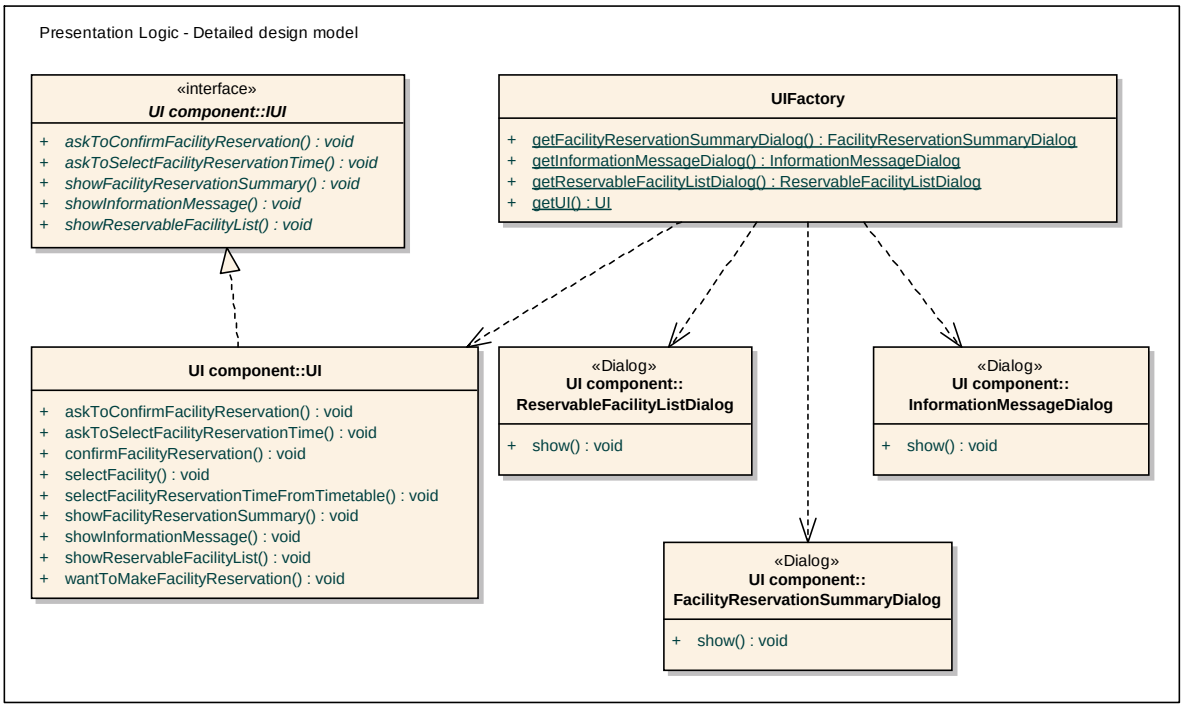

Figure 3.19: Example of a UI component's detailed design model

To be able to generate a dialog or frame a GUI meta-model is necessary that describes possible structures of dialogs and frames. The GUI meta-model is based on desktop application UI pattern. The next section describes this meta-model followed by an explanation of the generation of dialogs and frames out of RSL UI elements.

3.3.1 Dialog generation

The target of the second transformation step is a GUI model specifying the structure of dialogs and frames. The GUI model conforms to a GUI meta-model describing an abstract GUI toolkit. The GUI meta-model presented here is based on the Java Swing Model but is generalised in a way to be easily adaptable to other GUI toolkits like Windows Forms (e.g., we omit the "J" in front of each class name). It should be mentioned that the GUI meta-model presented here is not part of the RSL and the SCL.

The GUI meta-model provides a possible way for specifying the structure of dialogs and frames contained in the User Interface::UI component and the GUI, respectively. It organises all GUI widgets in the form of a tree as common to current GUI toolkits. The GUI meta-model is represented as a UML class diagram. It also supports the possibility to establish *traceability links*.

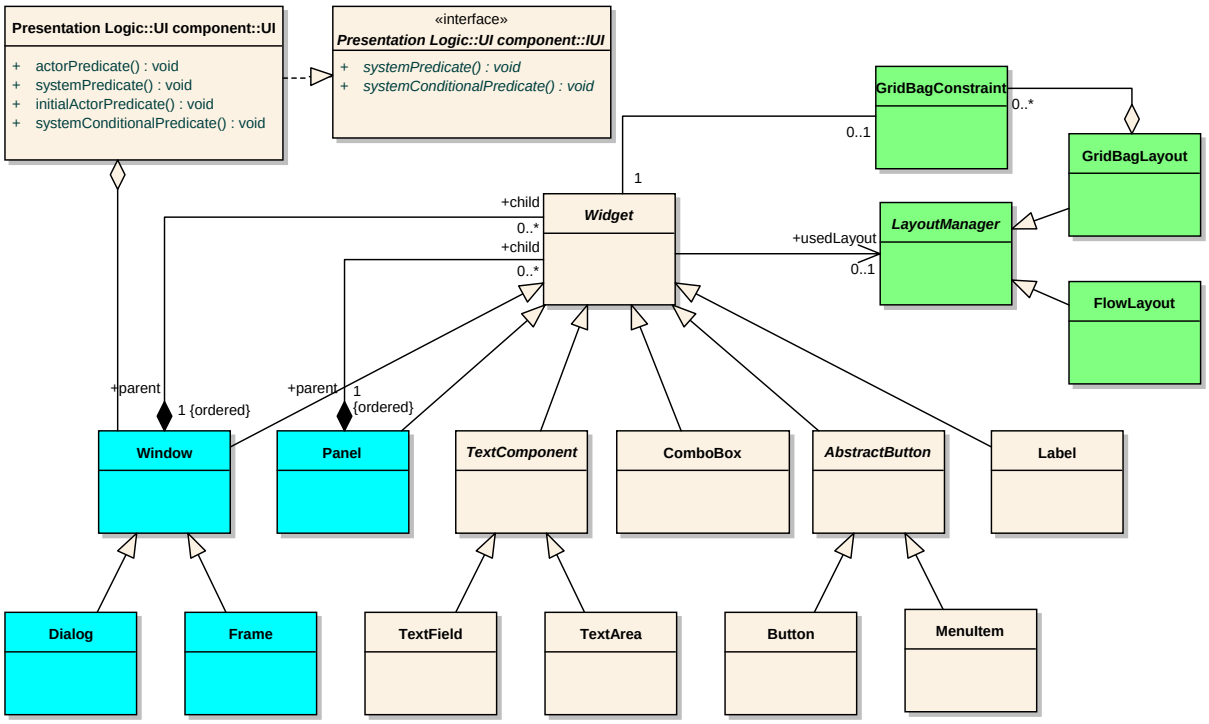

Figure 3.20: A conceptual GUI meta-model

Figure 3.20 shows the GUI meta-model relevant to generating the detailed design of dialogs. The classes and structure of the meta-model are mainly derived from Java Swing. The base class for all widgets is the abstract class Widget. The specialised classes Window, Dialog, Frame and Panel in the left part of Figure 3.20 depict GUI containers that aggregate other widgets. Since in Java Swing each component is also a container, the conceptual GUI meta-model presented here also does not explicitly distinguish between container and non-container elements. Conforming to Java Swing, all GUI widgets can be associated with a LayoutManager, but usually it makes sense to attach layout managers only to GUI containers. Some layout managers like GridBagLayout assign additional constraints (e.g., row, column, padding, etc.) to the GUI widgets aggregated in the associated GUI container. Other layout managers like FlowLayout only depend on the order the widgets are added to the container. This order is assured by the order constraint of the aggregation between panels or windows and their child widgets.

The content of a window (dialog or frame) is determined by the following rules. If a detailed specification of an RSL UI presentation unit is available, the following rules apply:

- Every RSL UI container is transformed into a Panel. By default, a FlowLayout is associated with the Panel. The name of the generated panel and flow layout will be the name of the UI container suffixed with “Panel” and “FlowLayout” respectively. All RSL UI elements contained in the RSL UI container get processed in the order defined by the associated RSL presentation order. The following rules are applied to the contained RSL

UI element depending on their specialisation type. Each generated widget is then added to the Panel.

- Every RSL input UI element is transformed into a TextField and an associated Label. If it is possible to determine the type of data through the associated SVO scenario sentence, it is possible to generate a text area, spinner, date/time picker, gauge or slider instead of a text field. The default text of the generated label will be the predicate of the SVO sentence. If the associated SVO sentence is an actor predicate, an input change method is generated, that calls the application logic method via the UI class. The name of the generated text field and label will be the name of the input UI element suffixed with "TextField" and "Label" respectively.

- Every selection UI element is transformed into either a ComboBox, ListBox, CheckBox or RadioButtons depending on the number of contained RSL option UI elements and on the allowance of multiple selection. For each RSL option UI element a single CheckBox or a single RadioButton is generated depending on the value of the isReSelectable attribute. Additionally, an associated Label is generated. The default text of the generated label will be the predicate of the SVO sentence. If the associated SVO sentence is an actor predicate, a selection change method is generated, that calls the application logic method via the UI class (e.g. SelectFacility). The name of the generated input widget will be the name of the input UI element suffixed with the type name. The name of the generated Label Widget will be the name of the label element suffixed with "Label".

- Every RSL trigger UI element associated with an RSL user action results in the corresponding specialisation of AbstractButton and an action method assigned to the button that triggers the actor predicate associated with the RSL user action. If the SVO sentence is an initial actor predicate, a MenuItem specialisation will be generated. The name of the generated specialisation of abstract button will be the name of the trigger UI element suffixed with "Button" or "MenuItem".

If no detailed specification of an RSL UI presentation unit is available, the following rules can be used to generate a first prototype for the detailed design model of the dialogs contained in the UI component. Those rules enforce the use of English in writing requirements. They can be a guideline for the designer. The detailed design model needs to be manually reworked by the designer:

- Every SVO sentence attached to an RSL UI scene where the subject refers to the system and the verb denotes a required selection by the user like "offer" is transformed into a Label representing the object and either a ComboBox, ListBox, CheckBox or RadioButtons depending on the number of contained data and if multiple selection is allowed or not.

- Every SVO sentence attached to an RSL UI scene where the subject refers to the system and the verb denotes gathering information from the user like "ask" or "request" is transformed into a Panel containing a list of pairs of Label and input widgets. These widgets represent the domain information referred to by the object of the sentence. The name of the domain element is transformed into the Label and the type of the domain element determines the type of the input widget. By default, the type of the input widget is TextField allowing to enter any kind of information.
- Every SVO sentence attached to an RSL UI scene where the subject refers to the system and the verb denotes the presentation of information like "inform" is transformed into a Label representing the information denoted by the object of the sentence.
- Every SVO sentence attached to an RSL user action where the subject refers to the user of the system results in the creation of a Button that allows submission of information that the user entered based on other SVO sentences attached to predecessor RSL UI scene of the RSL user action. If there is also a conditional sentence associated with the RSL user action, the condition evaluation is also included in the action method of the generated button.

Figure 3.21 shows an RSL storyboard for the "Make facility reservation" scenario used as an example throughout these sections. The storyboard consists of RSL UI scene, each attached with a list of SVO sentences and an RSL presentation unit displaying the envisioned dialog structure. Each RSL UI presentation unit gets transformed into dialogs according to the rules above. The generated dialog structure for the "Reservable Facility List Dialog" and the "Facility Reservation Summary Dialog" are shown in Figure 3.22 and Figure 3.23.

The ReservableFacilityListDialog in Figure 3.22 is composed of two Labels, two ComboBoxes and two Buttons. The "UI presentation order" is transformed to an instance of FlowLayout assigned the ReservableFacilityListDialog. The order is given by the order constraint of the aggregation specified in the GUI meta-model. The flow layout places everything in one row until there is not enough space in the row which leads to switching to the next row. In the reservable facility list dialog only the buttons are placed on the same line.

Figure 3.23 illustrates the GUI Model for the "Facility Reservation Summary Dialog", which results from the UI presentation unit shown in scene 4 in Figure3.21. The dialog is invoked by the "show facility reservation summary" system predicate. The FacilityReservationSummaryDialog is composed of a Label displaying the status and a Button arranged with a FlowLayout.

Figure 3.24 shows a typical sequence for the "Make facility reservation" scenario. The "wants to make facility reservation" initial actor predicate leads to the system predicate "show reserv-

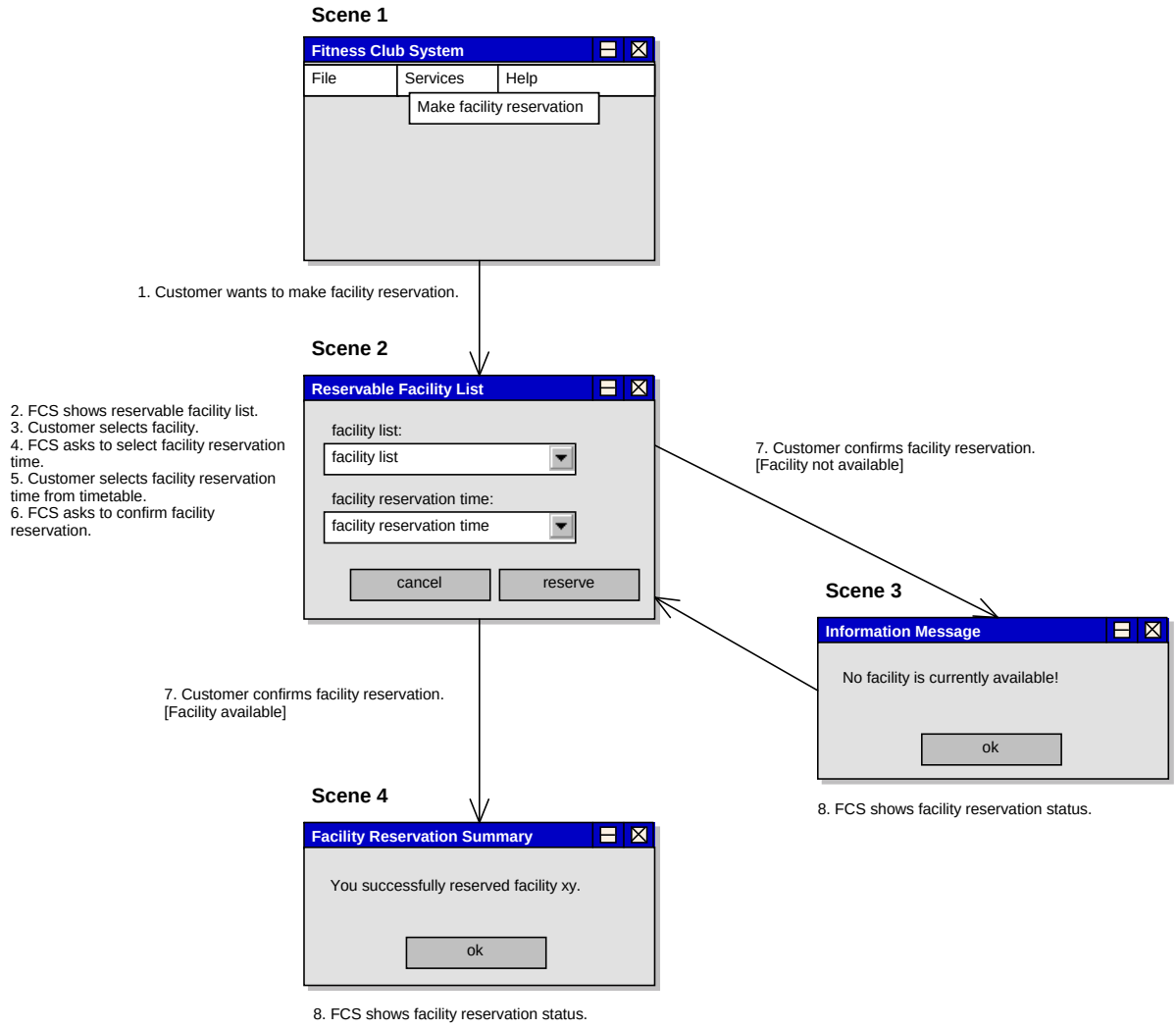

Figure 3.21: UI storyboard for “Make facility reservation” scenario

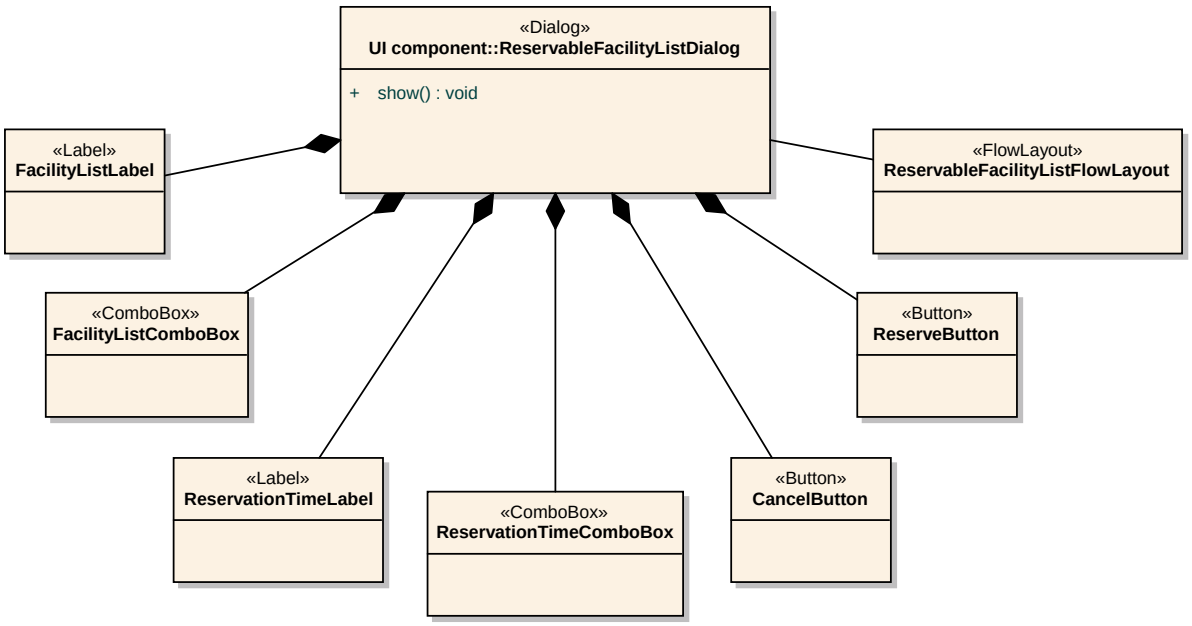

Figure 3.22: Reservable facility list dialog structure

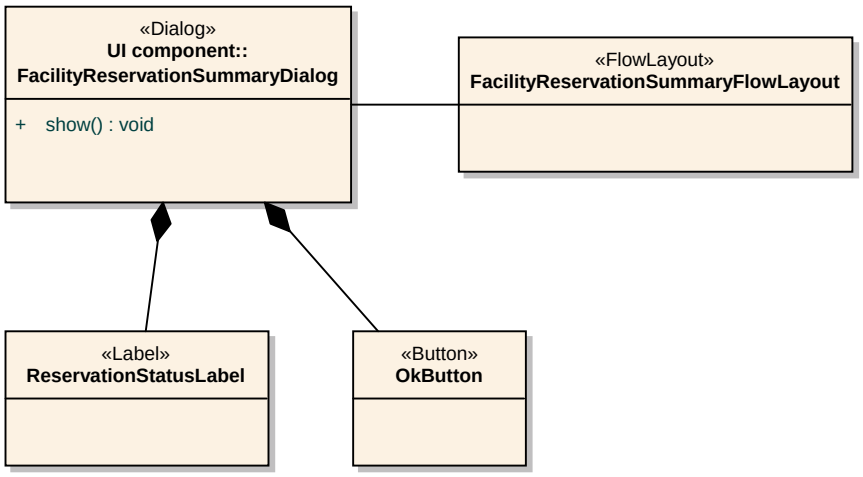

Figure 3.23: Facility reservation summary dialog

able facility list" which is transformed into a call on the UI class instance. This instance contacts the UI factory and retrieves the "ReservableFacilityListDialog" that is displayed to the user afterwards. The change of the "FacilityListComboBox" leads immediately to a method call corresponding to the "select facility" actor predicate. The following "ask to select facility reservation time from timetable" system predicate updates the selectable items of the "Facility ReservationTimeComboBox" according to the selected facility. After the user selects a reservation time, the OKButton gets enabled with the next system predicate "ask to confirm facility reservation". A click on the OKButton closes the dialog and sends the actor predicate "confirm facility reservation". This SVO sentence is associated with the corresponding RSL user action leading to the next scene in the UI storyboard.

3.4 Transformations to detailed design model

As one of main objectives of the MDA approach is to separate the specification of functionality from the specification of the implementation of that functionality on a specific technology platform [MM03], only after PIM architecture is ready (generated from requirements and enhanced by an architect) it can be transformed into PSM detailed design. Below a set of rules for transformations of every tier in 4-layer architecture is presented. The transformation process uses only information contained in the architectural model, assuming that transformation from requirements to architecture extracted all possible information for generating the detailed design model.

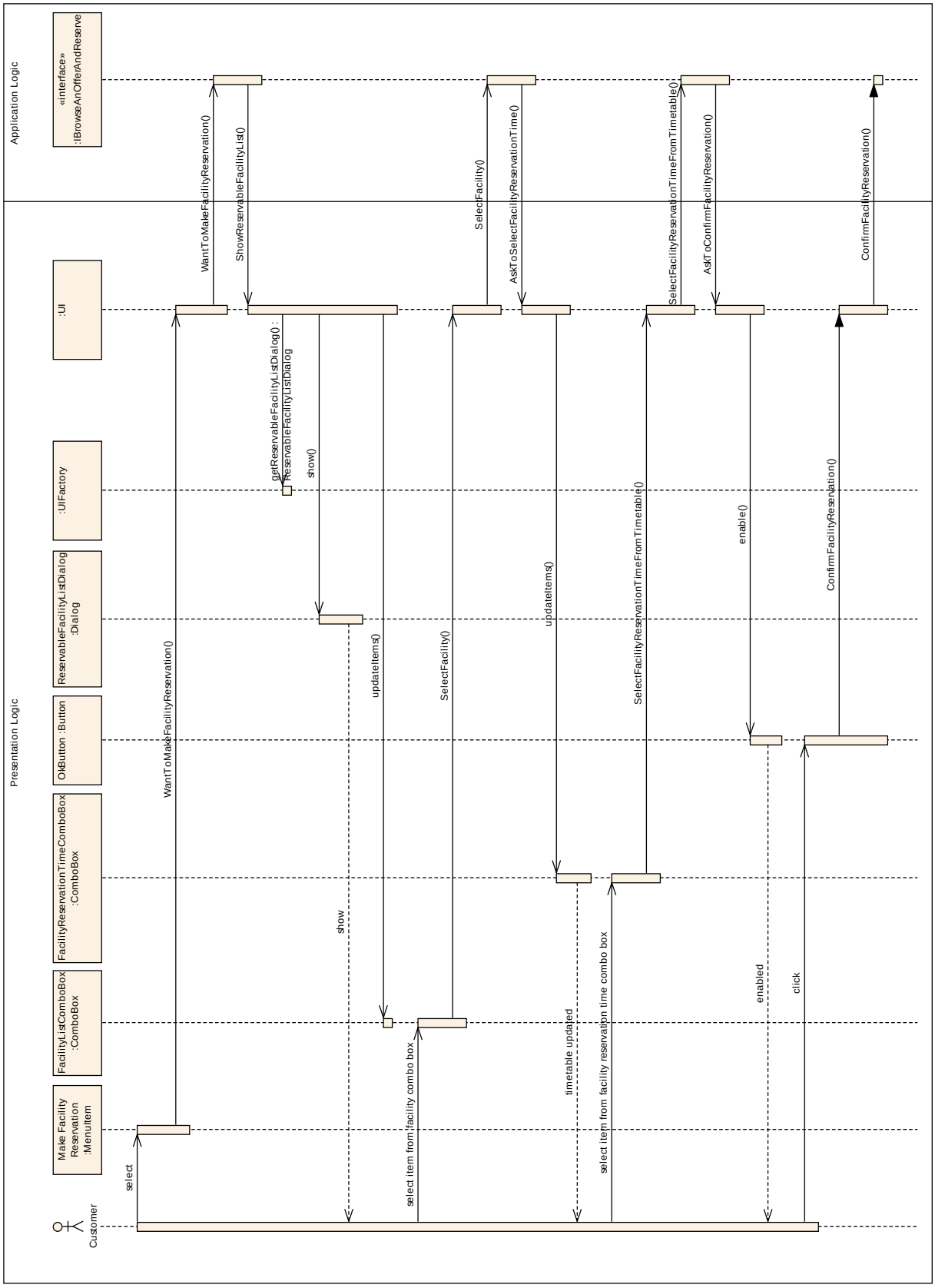

Figure 3.24: Customer wants to make facility reservation UI scenario.

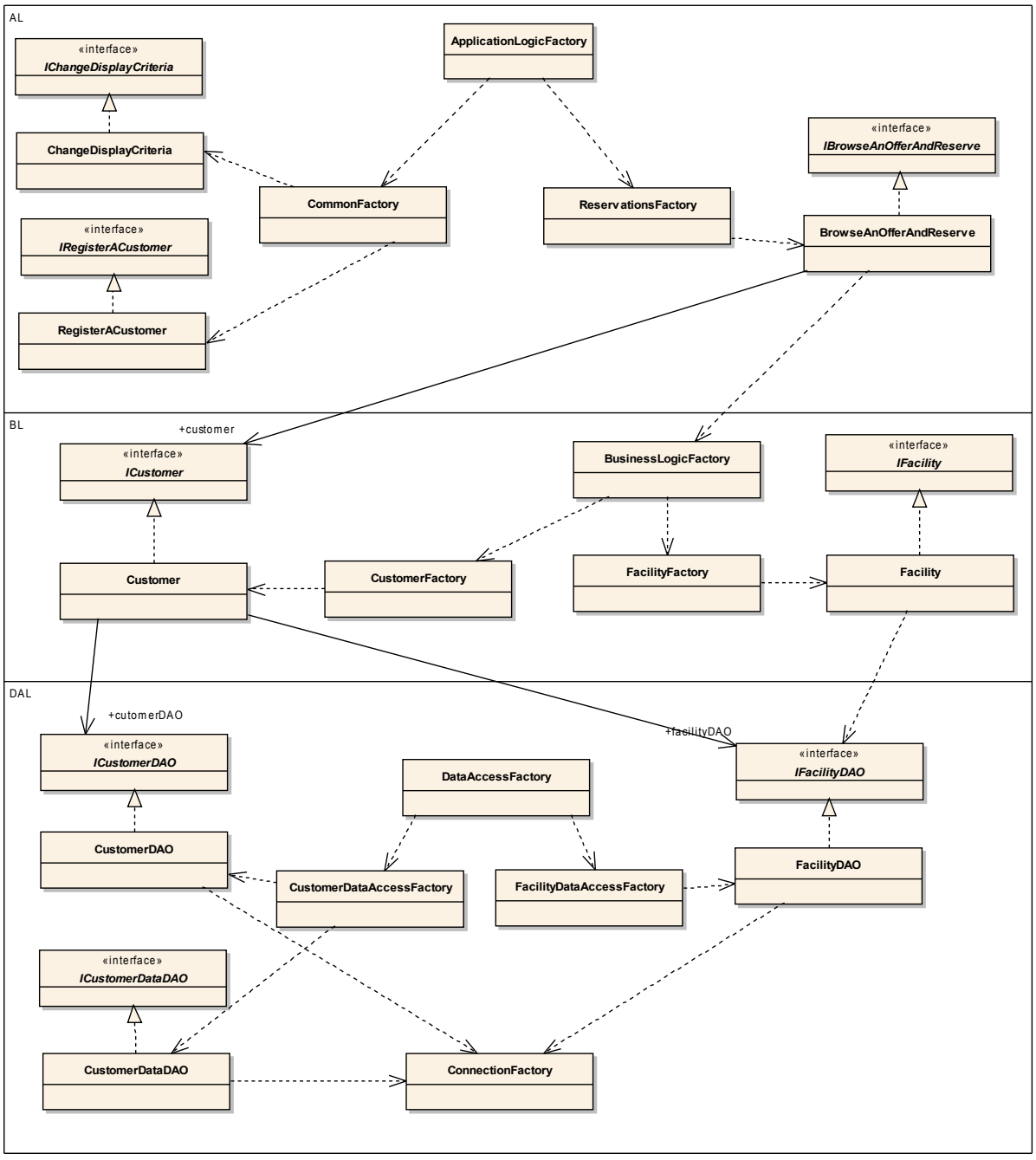

Figure 3.25: Overview of detailed design model generated from architectural model

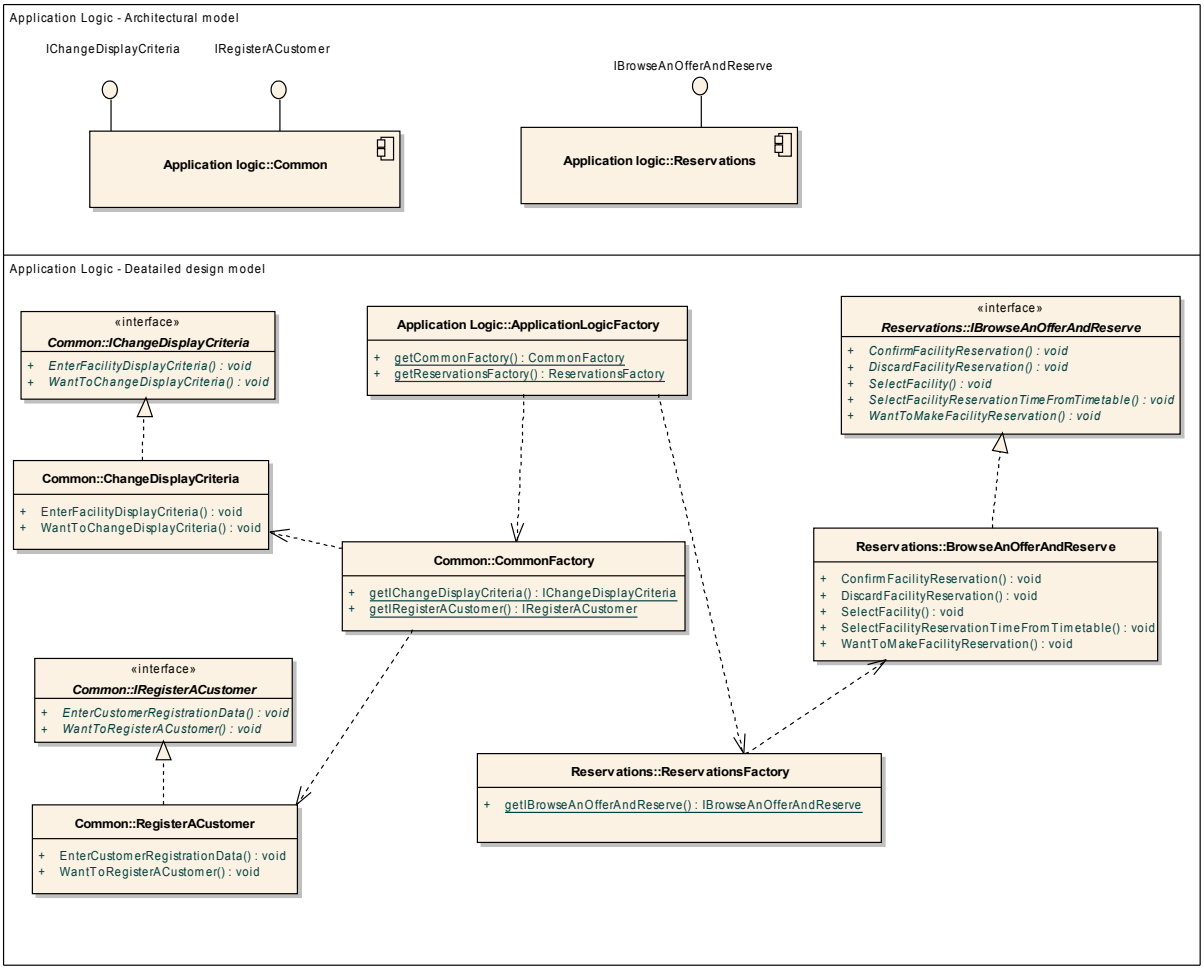

Figure 3.26: Generation of Application logic detailed design

3.4.1 Generation of Application Logic

Below we present rules for generation of the application logic layer elements, including their naming patterns and relationships that may exist among them.

- for the application logic layer one application logic factory is created. It is a static class with name “AppLogicFactory”.
  - for every component in the application logic layer a method returning a component factory is generated. The method name is composed of “get” prefix and component name (e.g. getReservationsFactory)
- for each component in the application logic layer a corresponding component factory is created. They are static classes with a name composed of a component name + “Factory” suffix (e.g. ReseservationsFactory)

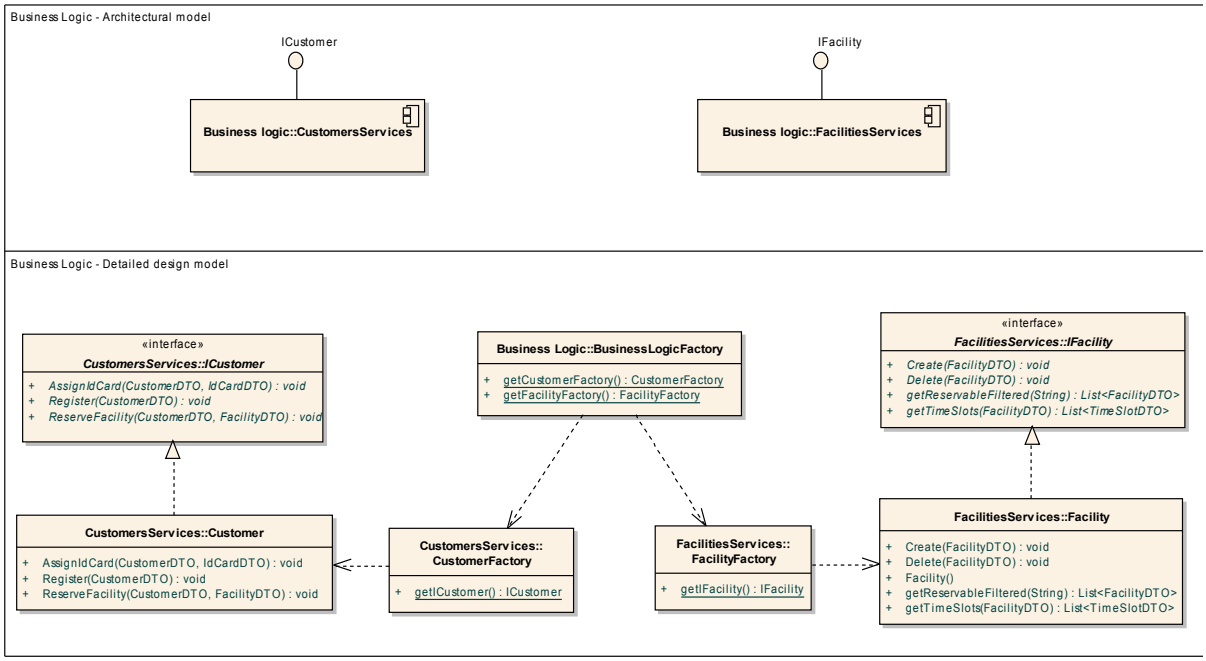

Figure 3.27: Generation of Business logic detailed design

  - for every interface in a given component a method returning a realisation of this interface is generated. The method name is composed of "get" prefix and interface name (e.g. getIBrowseAnOfferAndReserve)

- for each interface in the application logic layer a corresponding interface and implementation class are generated (with realisation relation)
  - interface methods are the same as these in architectural model
  - generated interface has the same name as in the architectural model (e.g. IBrowseAnOfferAndReserve)
  - generated implementation class has the same name as interface without "I" prefix (e.g. BrowseAnOfferAndReserve)
- for the application logic factory, dependencies to component's factories are generated
- for each component factory dependencies to implementation classes are generated. These dependencies point at each class implementing an interface of this component.

3.4.2 Generation of Business Logic

Below we present rules for generation of Business Logic layer elements, in a way similar to rules for application logic.

- for business logic layer one business logic factory is created. It is a static class with the name “BusinessLogicFactory”.
  - for every component in the business logic layer a method returning component factory is generated. The method name is composed of a “get” prefix and a component name (e.g. getCustomerServicesFactory)
- for each component in the business logic layer a corresponding component factory is created. They are static classes with the name composed of component name + “Factory” suffix (e.g. CustomerServicesFactory)
  - for every interface in a given component, method returning realisation of this interface is generated. The method name is composed of “get” prefix and interface name (e.g. getICustomer)
- for each interface in the business logic layer a corresponding interface and implementation class are generated (with realisation relation)
  - interface methods are the same as these in the architectural model
  - a generated interface has the same name as in the architectural model (e.g. ICustomer)
  - a generated implementation class has the name as interface without “I” prefix (e.g. Customer)
- for business logic factory dependencies to component’s factories are generated
- for each component factory dependencies to implementation classes are generated. These dependencies point at each class implementing an interface of this component.

3.4.3 Generation of Data Access Layer

The rules for the data access layer are analogous to the rules presented in previous sections.

- for the data access layer one data access factory is created. It is a static class with name “DataAccessFactory;”.
  - for every component in the data access layer a method returning component factory is generated. The method name is composed of a “get” prefix and a component name (e.g. getCustomerDataAccessFactory)

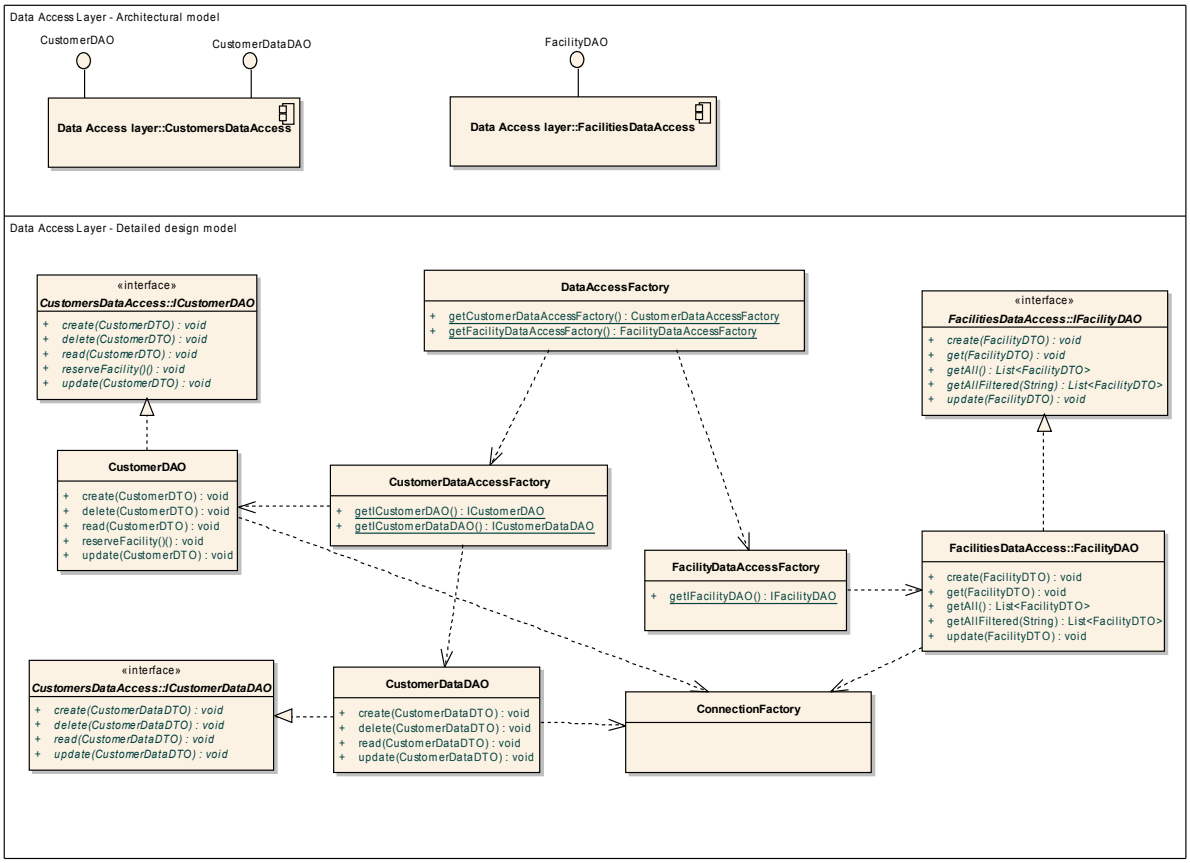

Figure 3.28: Generation of Data access layer detailed design

- for each component in the data access layer a corresponding component factory is created. They are static classes with a name composed of component name + "Factory" suffix (e.g. CustomerDataAccessFactory)
  - for every interface in a given component a method returning realisation of this interface is generated. The method name is composed of a "get" prefix and an interface name (e.g. getICustomerDataDAO)
- for the data access layer one connection factory is created. It is a static class with name "ConnectionFactory"
- for each interface in the data access layer a corresponding interface and implementation class are generated (with a realisation relation)
  - interface methods are the same as those in the architectural model
  - generated interfaces have the same name as in the architectural model (e.g. ICustomerDataDAO)
  - generated implementation classes have the same name as the interface without "I" prefix (e.g. CustomerDataDAO)
- for data access factory dependencies to component's factories are generated
- for each component factory dependencies to implementation classes are generated. These dependencies point at each class implementing an interface of this component.

3.4.4 Generating data transfer objects (DTOs)

All DTOs from the architectural model are copied with associations between them to the detailed design model

3.4.5 Generating relationships between layers

Associations between all layers are generated based on sequence diagrams. Messages between elements of the application logic layer and business logic layer are generated automatically from requirements. Messages between elements of the business logic layer and the data access layer are not generated during transformation from requirements and can be set manually by an architect.

Relationships between the application logic layer and the business logic layer

- for each class implementing an interface in the application logic layer, which uses the business logic layer, a dependency to the business logic factory is generated
- each message between the application logic layer interface and the business component interface in architectural model is transformed into an association between implementation classes in the application logic layer and the business interface in the detailed design model. The source of the association is a class implementing interface which is the source in the sequence diagram.

Relationships between the business logic layer and the data access layer

- for each class implementing interface in the application logic layer, which uses the data access layer, dependency to the data access factory is generated
- each message between the business logic layer interface and the data access layer interface in the architectural model is transformed into an association between the implementation class in the business logic layer and the data access interface in the detailed design model. The source of association is a class implementing interface which is the source in the sequence diagram.

Chapter 4

Formal definition of transformations in MOLA

This chapter describes the MOLA implementations of transformations described informally in the previous chapter. Again, the content of this chapter is a further development of ideas from deliverable 3.2.1, in order to obtain a practically usable set of transformations for experimental evaluation of the approach in the next deliverables.

The informal transformations in the chapter 3 are based on one of the possible solutions for software case architecture in ReDSeeDS. It is the 4-layer software architecture model introduced already in deliverable 3.2.1. Chapter 3 of this deliverable refines further this model. In particular, sufficient details are added to the detailed design model.

Precise transformations defined in MOLA can be applied to all design steps of a software case in ReDSeeDS, wherever some automatic transition from one model to another is possible. However, the value and applicability of transformations significantly depend on the informal transformation algorithms and principles which have been designed for that step. These algorithms must reflect natural design dependencies of some model elements upon the corresponding fragment of the previous model. Therefore MOLA transformations in this deliverable have been built only for those steps, where approved informal algorithms exist.

The most important and also the most elaborated currently are transformations from the Requirements model to the Architecture model, applicable to the above mentioned 4-layer architecture. They are a further refinement of the informal transformations provided as an example in deliverable 3.2.1. According to this, the corresponding MOLA transformations are also essential improvements of the MOLA transformation example in section 6.5 of deliverable 3.2.1.

In addition to activity representations, textual representations of use cases by SVO sentence scenarios can also be used as a source for these transformations. However, the most important fact is that these MOLA procedures now represent complete descriptions of transformations, without any omissions. The new features of informal algorithms, such as the generation of dependencies between components and interfaces and associations between DTO classes, are also implemented in the transformation. Transformations are also partially tested now on some test model instances, using the initial version of MOLA tool execution facilities. The main goal of transformation development in this deliverable is to provide solutions, which could be used as a base for testing the whole approach on real examples within the next deliverables. Certainly, some updates may be required, when the first prototype of the RSL tool really appears. These improved transformations from RSL to Architecture model are described in detail in section 4.2.

Chapter 3 in this deliverable now provides also an initial version of informal transformations from the architecture model to detailed design model. They are defined in the context of the same proposed 4-layer software architecture model. The architecture model is assumed to be refined manually by a software architect, after an initial version of it was generated by transformations from requirements. Some of these refinements are crucial for these transformations. The static structure of the detailed design model (actually, a set of class diagrams) to a significant degree can be generated from the refined architecture model. The corresponding informal algorithms in chapter 3 provide a clear picture of these transformations, sufficient for implementation, though they are more complicated than for transition to the architecture model. According to the provided algorithms, an initial version of the respective MOLA transformations is given in section 4.4. Since the informal descriptions of the algorithms may not be completely stable yet, the corresponding MOLA procedures are not complete in this deliverable, only the most essential solutions are provided. These transformations differ from the above ones by the fact that both source and target models are UML based.

This chapter concludes with a brief sketch of other situations, where transformations in MOLA would be required for the support of building a software case. In particular, problems of code skeleton generation are briefly sketched.

Thus this chapter describes a possible set of transformations which is required for support of software case building according to a specific software architecture model - the 4-layer software architecture model. Certainly, if this model changes both the informal algorithms and formal transformations in MOLA must be adapted. However, the basic transformation principles would remain the same. The implemented transformations are adapted to support integration to software case repository, details of this adaptation are described in chapter 5.

One more issue not addressed in this deliverable is the update versions of transformations. Currently only the initial version of the corresponding target model is generated. The update version of transformations must be aware of possible manual extensions of the existing part of the target model, which must not be overwritten. The structure of transformation implementation in MOLA and the generated traceability links are sufficient to incorporate also this aspect. However, the corresponding update branches have not been implemented yet, because it would increase the size of transformations, but no new solutions would be required. Their implementation is delayed till the next deliverable, when some initial RSL tool would really be available, because some modifications of algorithms are possible in relation to this.

4.1 Source and target metamodels

Each MOLA transformation must have a precisely defined source and target metamodels, which must be specified in accordance to MOLA MOF requirements, which have been described already in deliverables 3.1 and 3.2.1. If a chain of transformations is to be defined, then the target model (and metamodel, respectively) becomes the source for the next transformation in the chain. Therefore metamodels in this section are structured according to the language they define. Section 4.1.1 describes the used version of RSL metamodel. Section 4.1.2 describes a metamodel for a subset of UML, which is used both as a source and target metamodel for respective transformations.

4.1.1 RSL metamodel for transformations

As it was already mentioned, MOLA MOF requires some restrictions to meta modelling, basically, those of EMOF [Obj06] and in addition, currently single inheritance only. The previous deliverable 3.2.1, in its section 6.5.1, already contained such an adaptation of the relevant subset of the general tool-ready metamodel for RSL. However, the tool-ready metamodel for RSL itself has been slightly updated since then, to remove the found design errors. On the other hand, the chosen subset of RSL has been extended too, to include also the textual representation of use cases. Therefore the MOLA-ready metamodel for the relevant RSL subset must be repeated in this deliverable, in order to make the transformations readable.

The RSL subset currently usable for transformations includes use cases and their “formal” representations (activity and SVO sentences), use case packaging and relations, notions, their grouping and relations and, finally, phrase and term related elements which define the textual structure of the previously mentioned RSL elements. The selected subset of the tool-ready RSL

metamodel was converted according to EMOF requirements and multiple inheritance (actively used in the tool-ready RSL) was excluded. During that process those abstract superclasses, which provide no new attributes and whose associations are redefined at a lower level, have been simply eliminated. In addition, to ease writing of MOLA rules, role names have been added to many empty association ends in the original metamodel.

The provided MOLA-ready metamodel is semantically equivalent to the current version of the tool-ready RSL metamodel for the chosen RSL subset. The equivalence is at the instance level - if a software case repository is consistent to the tool-ready RSL metamodel, then instances of classes and associations in the subset can be exported to/from MOLA repository without any loss of information (see details in chapter 5). However, small modifications to this metamodel may be required when RSL tools become available within workpackage 5.

Similarly to the version in deliverable 3.2.1, the MOLA-ready RSL metamodel is represented by several class diagrams (figures 4.1 to 4.4). Each diagram shows related classes in some of the RSL packages.

The current version of the metamodel contains also temporary updates, which are necessary for transformations. First, the *name* attribute has been added to RequirementRepresentation class, in order to have names for different scenarios within a use case. Second, a string-typed attribute tag is added to RSLActivityEdge class, in order to have a place for a requirements designer to comment, which outgoing edge in branch node corresponds to which path (or scenario). Both these are issues, which must be solved in the context of an RSL tool, but since there are no clear visions of this tool yet, temporary solutions are offered.

4.1.2 Metamodel of UML subset

As it was already mentioned, the metamodel for a UML subset may play several roles for transformations. It is the target metamodel for transformations from RSL to architecture model. For transformations from architecture to detailed design model it is both source and target metamodel. Therefore this metamodel has been significantly extended with respect to its previous form in deliverable 3.2.1, though the approach is the same.

Several decisions have been made in relation to the used subset of UML and its version, in order to harmonise it with requirements from other possible ReDSeeDS tool components and with design styles used in the selected design architecture.

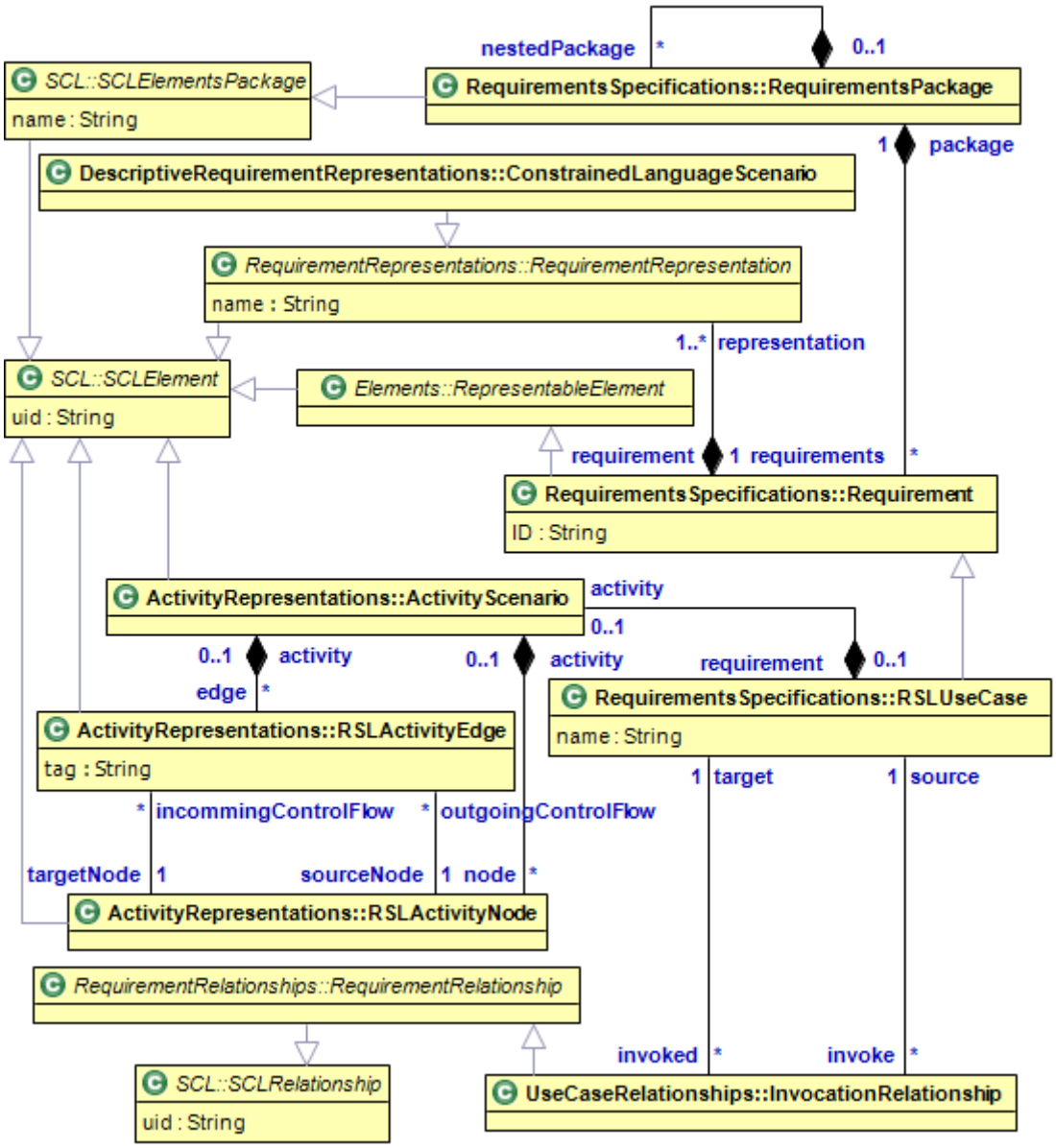

Figure 4.1: Top RSL metaclasses in MOLA-ready metamodel

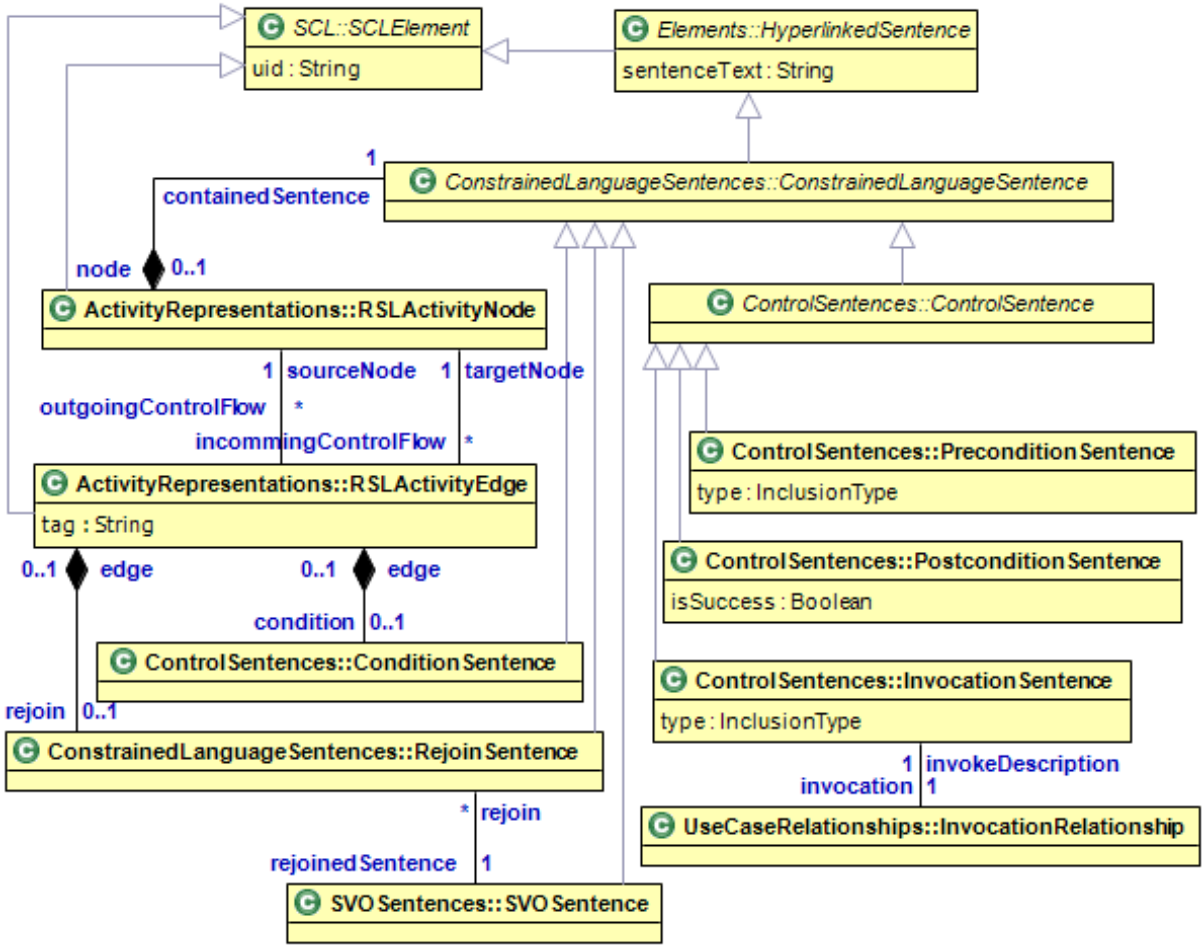

Figure 4.2: Constrained language sentences for activity scenario elements

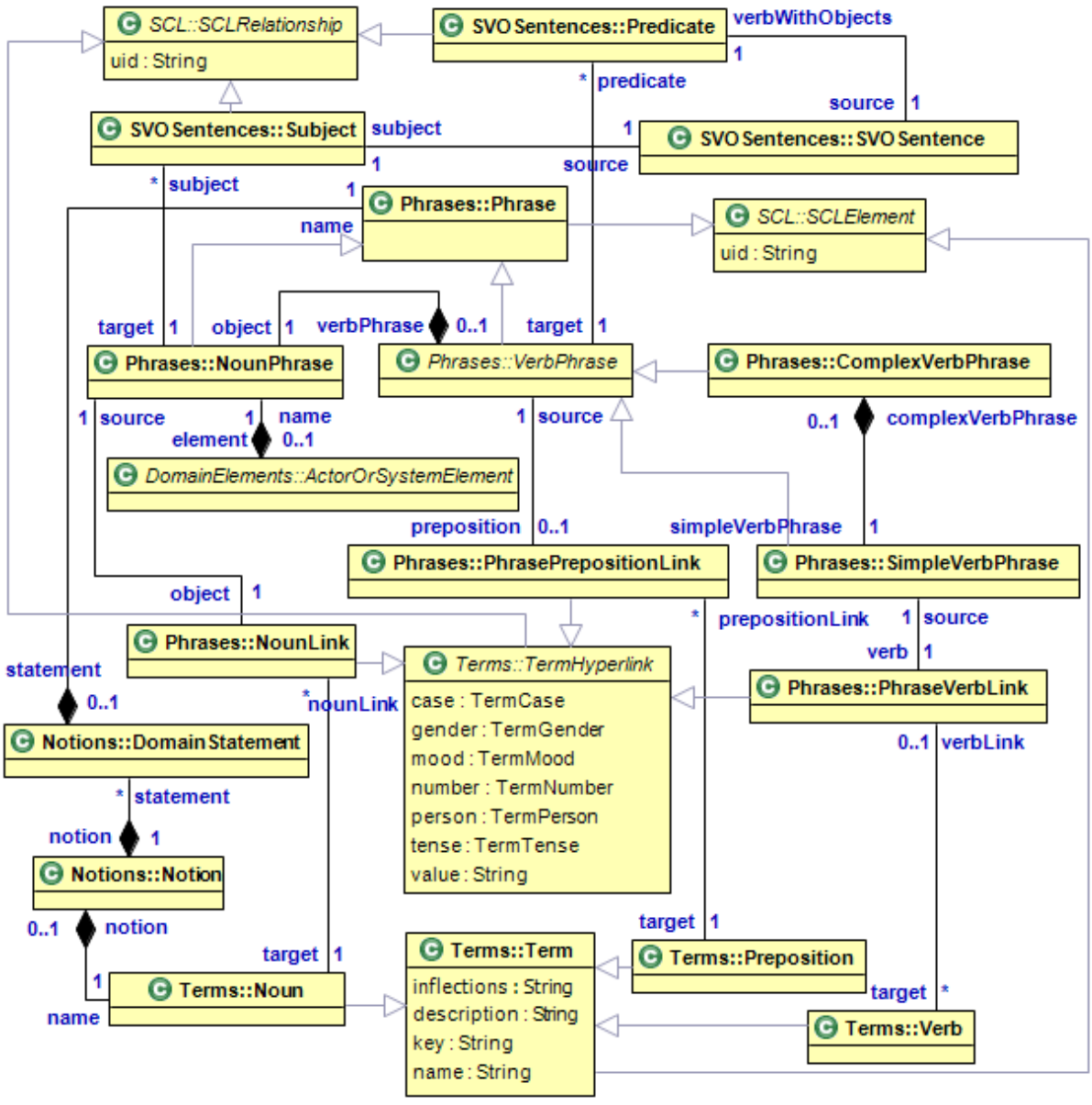

Figure 4.3: Phrases and terms used in MOLA-ready metamodel

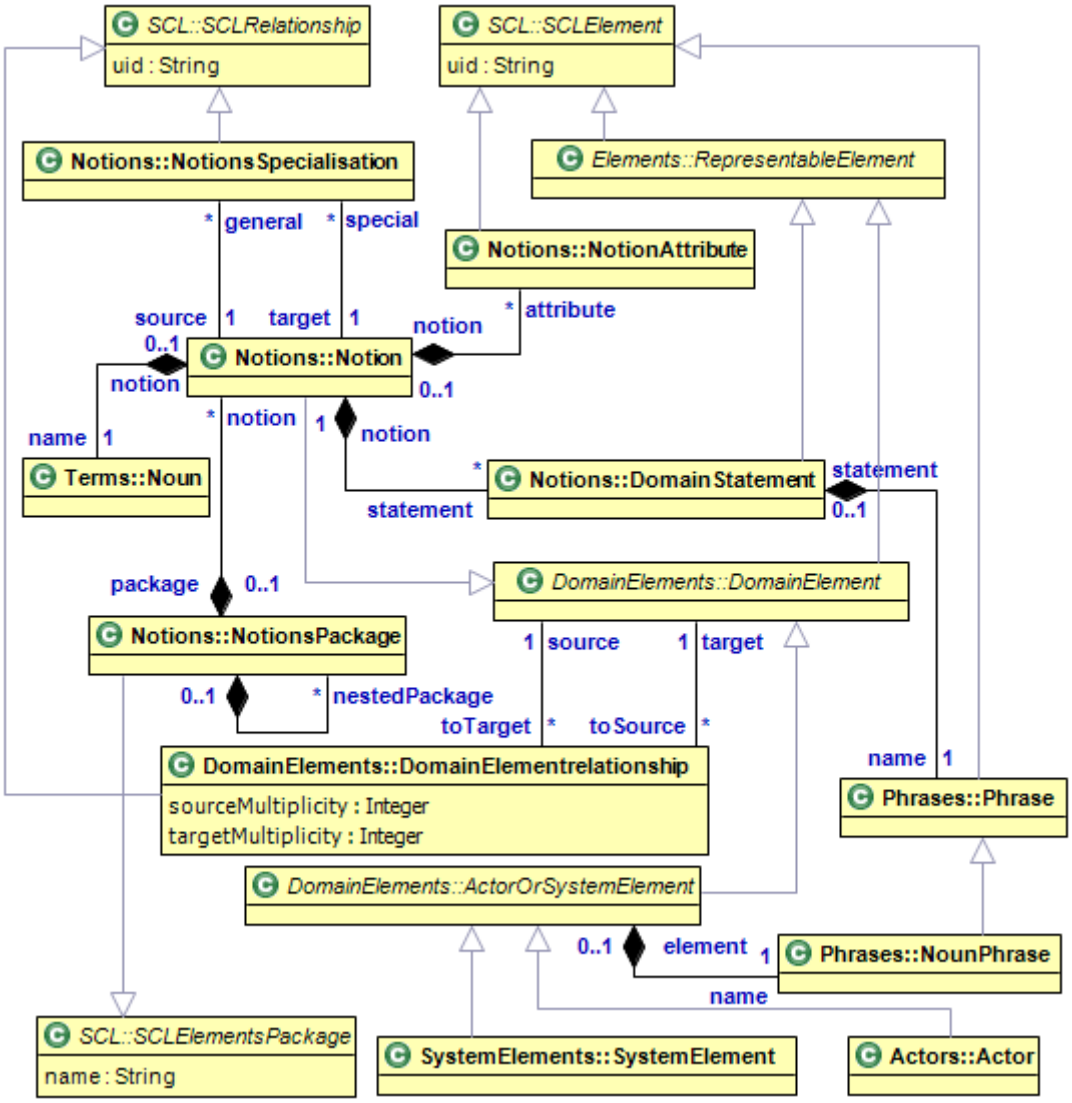

Figure 4.4: Notions and their relationships

Firstly, UML 2.0 is used as the base version of the metamodel. However, a relatively small subset is selected, containing only the basic UML notations typically used for software design specification. This is done to make the transformation writing in MOLA manageable. On the other hand, this subset is sufficient for the design style, proposed for design examples in ReDSeeDS.

The original UML 2.0 metamodel [Obj05] heavily uses CMOF [Obj06] elements. Therefore it had to be transformed in a way similar to the RSL metamodel, to make it MOLA-ready. It should be noted, that a similar approach is used internally for several professional UML tools, including RSA by IBM Rational [SCG+05]. There UML 2.0 metamodel is reduced to EMOF and single inheritance too.

The static structure part of UML includes support for all elements typically used in class diagrams: classes, interfaces, associations, data types etc. They are mainly contained in the Kernel package. The main elements for component modelling are also included (from the package BasicComponents). Since components in the proposed ReDSeeDS software architecture are used in a quite restricted way, UML elements meant for description of deployment components currently are not included. The subset of UML 2.0 elements for static structure modelling is included in a way which is semantically equivalent (instance equivalent) to the original UML 2.0 metamodel. Therefore it will be possible to use the original UML 2.0 metamodel for the software case repository definition and have a complete data-preserving import/export to MOLA repository. If in the future it would occur that more UML elements would be used in transformations then the metamodel for the subset is easily extendable to required features, without the need to modify the existing part. The only element which has been simplified with respect to the UML 2.0 standard, is a simplified stereotype concept, applicable to any UML element (without profiles and profile application, the same way as it is in Enterprise Architect and many other tools). Figures 4.5 and 4.6 show the static structure part of the used metamodel.

The situation with behaviour description is more complicated. The main requirement is to support relatively complete sequence diagram notation. The solution is again based on the preliminary solution in deliverable 3.2.1, however, a much wider set of sequence diagram features are supported. The basic metamodel is close to UML 2.0 (its Interactions package), but with some parts simplified. This is due to the fact of excessive complexity of Interactions package (already discussed in deliverable 3.2.1), which would make the transformations unnecessary complicated, without adding any modelling power. These excessive metamodel elements also could not be transferred to Enterprise Architect in a reasonable way since it uses a custom metamodel. Again, it should be noted, that the used solution is similar to that used in IBM Rational RSA [SCG+05]. A special situation is with conditional messages. All existing sequence diagrams in ReDSeeDS examples use the traditional UML 1.4 guard notation, but not the UML

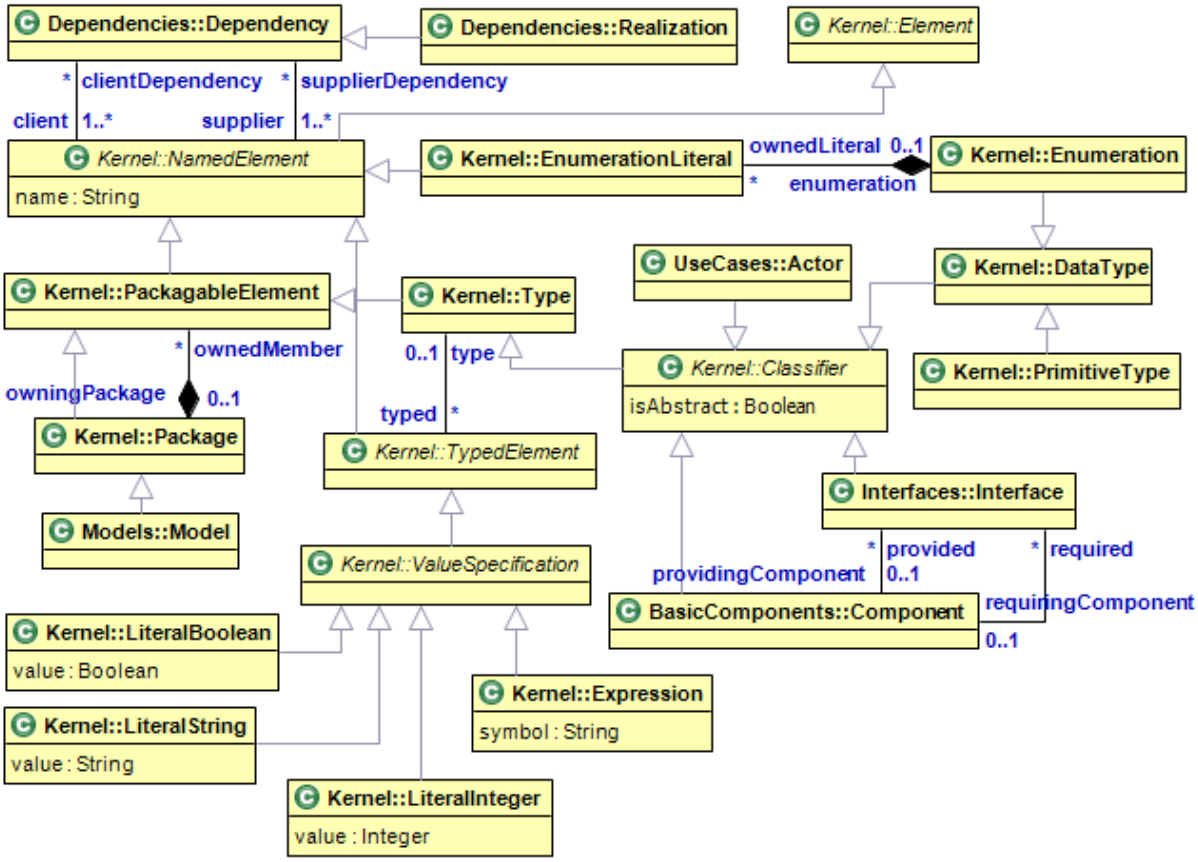

Figure 4.5: The used fragment of UML Kernel package

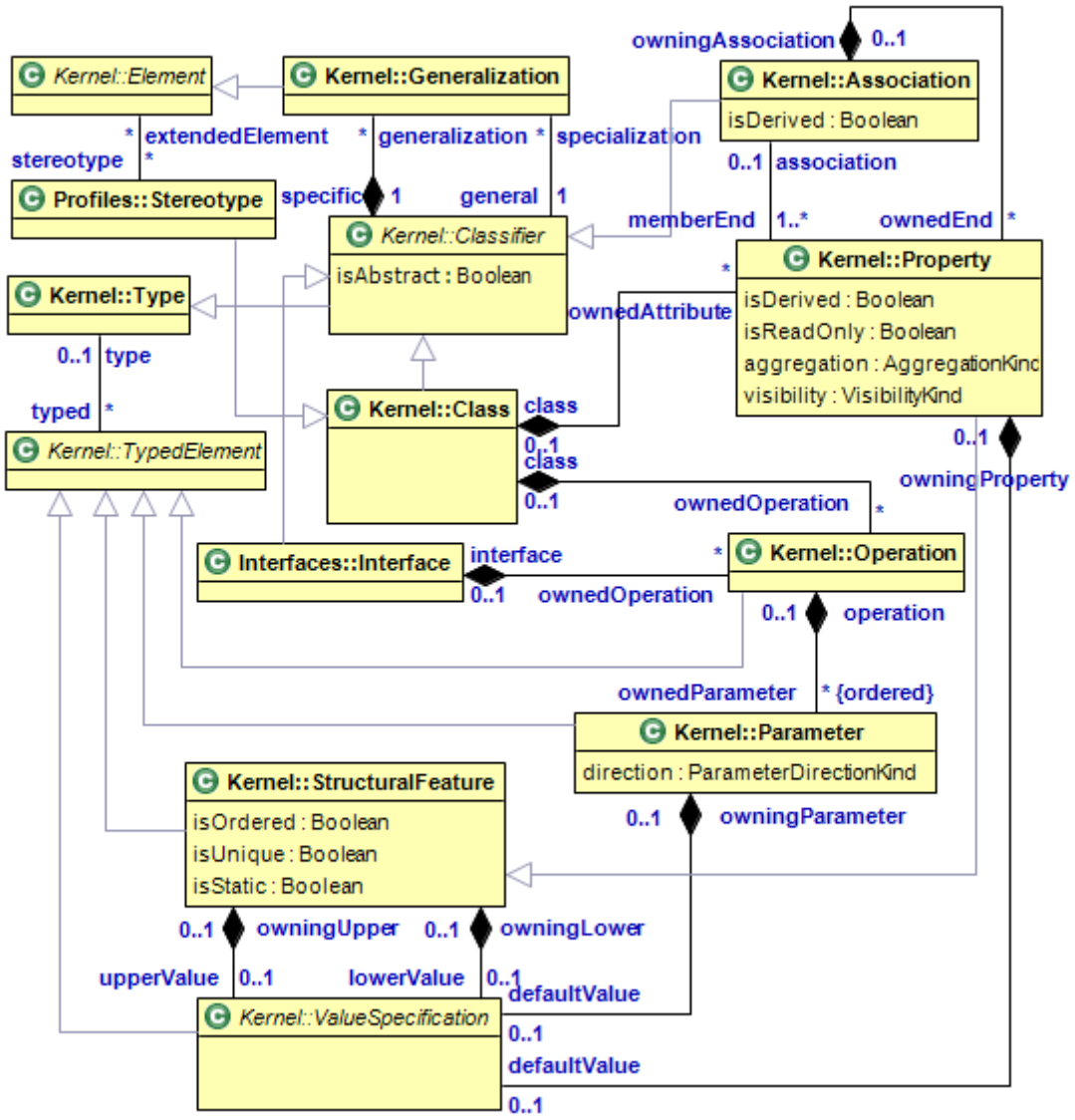

Figure 4.6: Metamodel fragment for classes

2.0 fragment notation. This is also easily possible in Enterprise Architect. Therefore the metamodel uses the UML 1.4 style solution for conditional messages. If however in future it would be decided to use true UML 2.0 fragments, they can easily be incorporated in the simplified metamodel. Figure 4.7 shows the interaction part of the metamodel. It should be noted that the repository metamodel also should be simplified in its Interactions package in a similar way, in order to have instance compatibility for transferring interaction-related data.

For traceability feature definition and their use as mapping elements for MOLA transformations, the general SCL solution already approved in deliverable 3.2.1 is again used the same way. It should be noted that it will also be the base for later incorporation of model update transformations, already mentioned above.

4.2 Transformations from RSL to architecture model

The description of informal transformations from RSL to architecture model in chapter 3 is only a slight improvement of the same algorithms in deliverable 3.2.1. They rely on the same proposed 4-layer software architecture model. Therefore the algorithms to be implemented in MOLA are nearly the same.

As it was already stated in the introductory part of this chapter, the main difference is a significantly more complete and practically usable implementation in MOLA. The transformations described in this deliverable may be used for practical evaluation of the approach, if facilities for data exchange between potential RSL tool repository and MOLA repository are implemented too (see chapter 5).

The following features (missing in deliverable 3.2.1) have been implemented now:

- representation of use cases also by textual SVO scenarios
- allowing several edges to exit an activity node and to build several sequence diagrams in the architecture model from one activity scenario accordingly (this construct is present in model examples, but was not supported by transformations in 3.2.1)
- the use of *rejoin* construct in scenarios
- complete implementation of invocation of another use case
- implementation of all situations for sequences of consecutive SVO sentences with respect to their subjects, relying on *recipient* association where appropriate

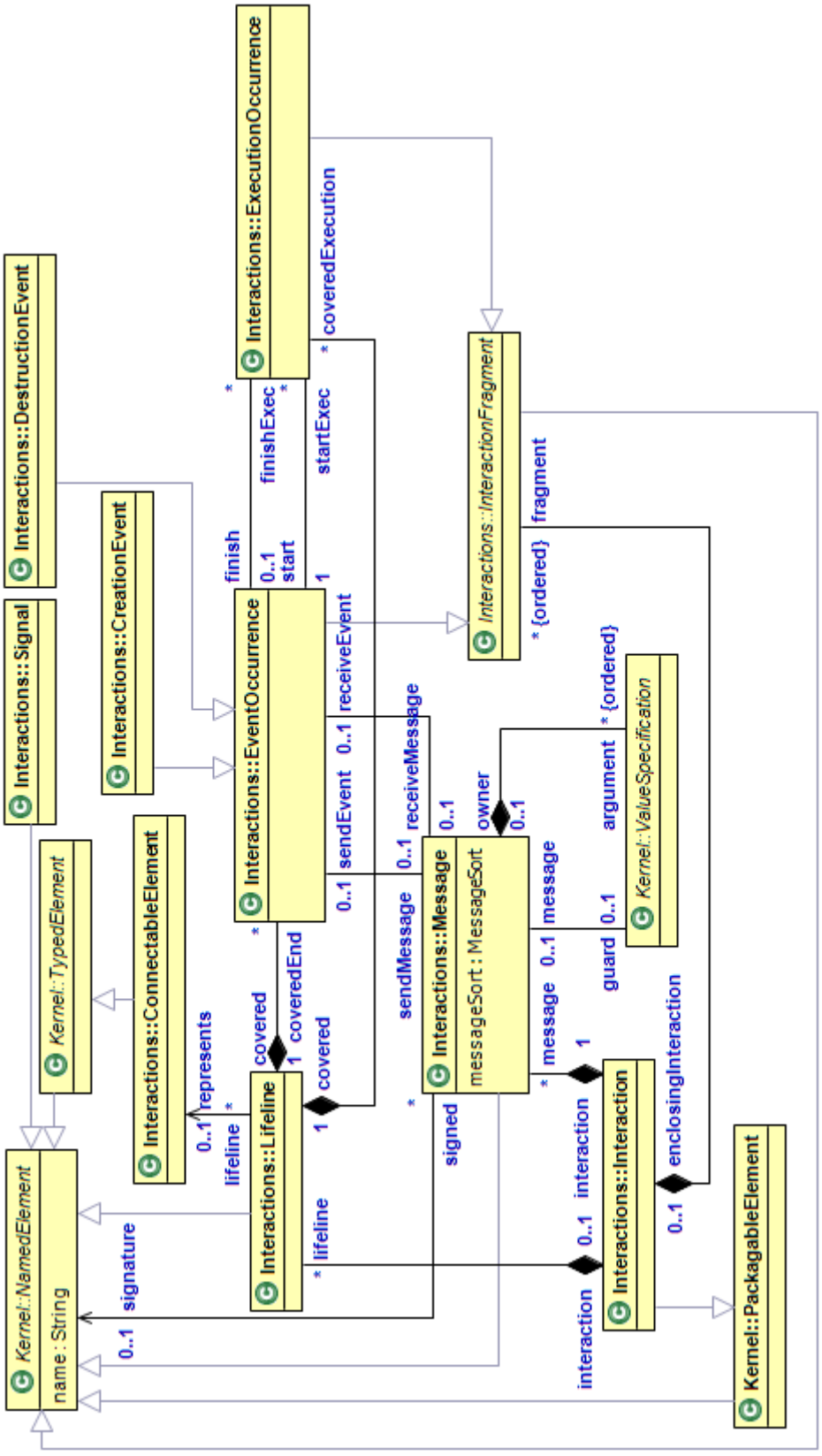

Figure 4.7: The simplified Interactions package

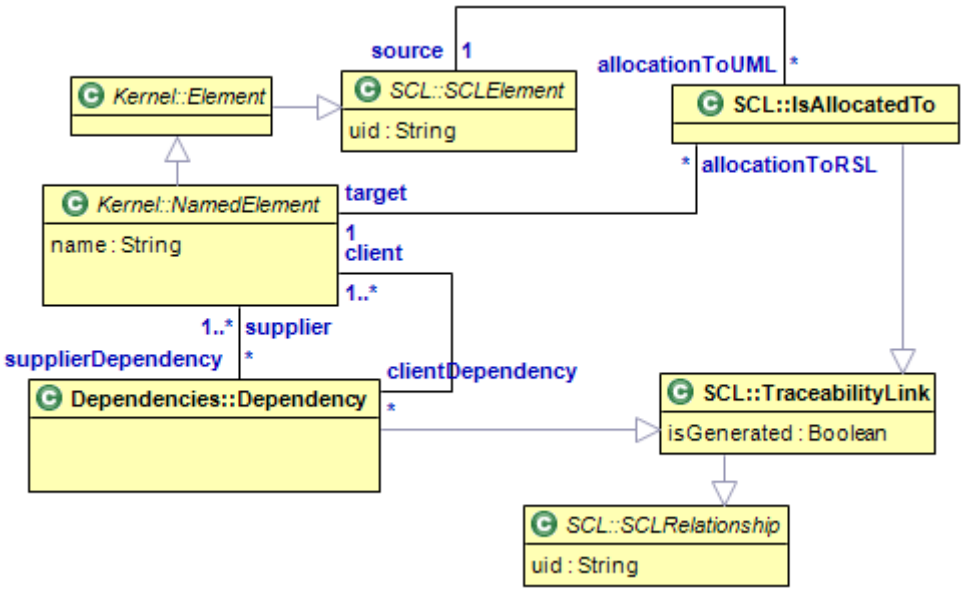

Figure 4.8: Traceability elements used for transformations

- implementation of some missing purely technical features, such as unnecessary instance filtering in the generated static structure.

The basic structure of the transformation is the same as in section 6.5 of deliverable 3.2.1. In particular, the building of static structure of the architecture model is nearly the same as in the previous version (deliverable 3.2.1). The main extensions are in procedures building the behaviour aspects.

MOLA procedures, which are nearly the same will be given in this deliverable without much comments. In general, the comments will be mainly related to algorithm details, but not MOLA constructs. Section 6.5 of deliverable 3.2.1 already has played the role of an additional MOLA tutorial.

Similarly to the version in deliverable 3.2.1, the transformation consists of two parts - processing the static structure and behaviour. The static structure building is practically the same as in the previous version. It consists of procedures stc_Structure 4.10, stc_CreatePackages 4.11, stc_ApplicationLogic 4.12, stc_DataTransferObject 4.13, stc_Actors 4.14, stc_DataAccessLayer 4.15 and stc_BusinessLogic 4.16. The procedures stc_DataTransferObject and stc_DataAccessLayer have been modified to generate DTO and DAO objects only for those notions, which appear in SVO sentences in the role of objects. The procedure stc_BusinessLogic still generates interfaces for all notions, but the unused ones (those which have no operations generated) are deleted at the end of transformation (by a new procedure flt_FilterServices 4.35). The dependencies between the UIComponent and application logic layer interfaces are generated at the static

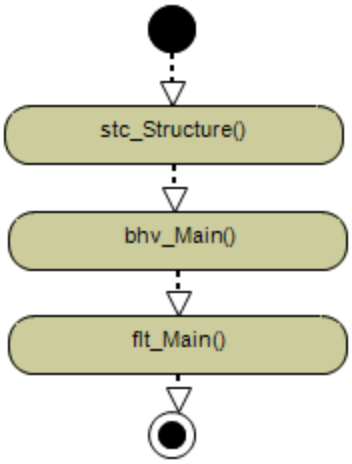

Figure 4.9: Main procedure of the transformation

structure step. Notion relationships are now analysed in the procedure stc_DataTransferObject 4.13. For each relationship a UML association is built. There currently is a problem in the RSL metamodel, that role names cannot be provided for relationships, therefore associations will be nameless now (this problem should be solved in the context of RSL tool).

The behaviour part is modified in several places due to the new functionality described above. Firstly, it is the possibility to represent a use case behaviour either by activity scenario or constrained language scenario (if both are present, the former one is used). To implement this, the main procedure for behaviour processing - bhv_SequenceDiagramms is modified in order to support also text scenarios.

The processing of constrained language scenarios is similar to the processing of activity scenarios, in the sense that an activity scenario is interpreted as a sequence of SVO or control sentences according to the sequence of nodes linked by edges, but a constrained language scenario already is such a sequence. If an edge in the activity scenario has a condition sentence, then the same condition sentence is inserted between the normal sentences in the constrained language scenario. A constrained language scenario can have no branching, therefore one sequence diagram is always generated from such a scenario. Otherwise the message building process is the same, based on subject analysis of two consecutive SVO sentences, the processing of control sentences is also the same.

Therefore the MOLA procedures for processing constrained language scenarios can be well integrated with the processing of activity scenarios. Actually, procedures are slightly redefined, so that after a common initial part processing a valid use case, the kind of its representation is found and processed accordingly. However, the differences are only in the way how one or more interaction (sequence diagram) instances are created and in the way how the current and the next

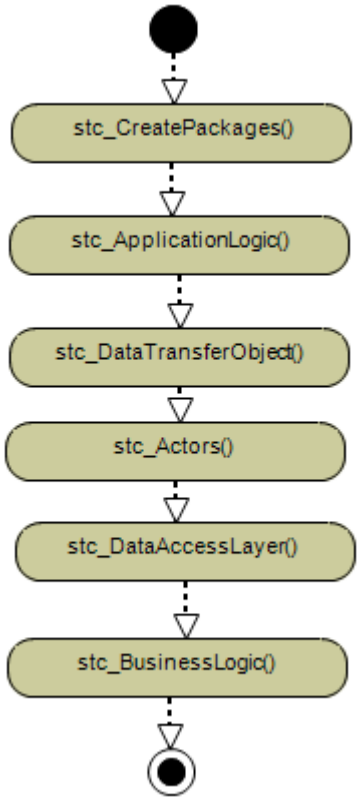

Figure 4.10: Main procedure for building the static structure

constrained language sentence (SVO or control sentence) is found in the scenario. The new feature that an activity scenario may contain branching nodes, and consequently, several paths corresponding to several interactions to be generated, is supported by a recursive procedure at this level. The proper processing of sentences is common to both cases and actually is the same as before.

Now, some more details of the restructured procedures.

A new important procedure for the behaviour part now is bhv_CreateUseCaseInteractions 4.19, which analyses a use case and finds, whether it has an activity scenario or constrained language scenario and decides, which kind of processing is required (if both are available, activity is used).

In the case of activity scenario, the procedure bhv_CreateActivityScenarioInteraction 4.20 is invoked for the given use case. It prepares the environment for a recursive traversal of the complete activity graph. First, the initialiser procedure bhv_CreateEmptyInteraction 4.21 is invoked, which creates an interaction with the static lifelines valid for all situations, including the lifeline for the current application layer interface (for this use case). Then the recursive

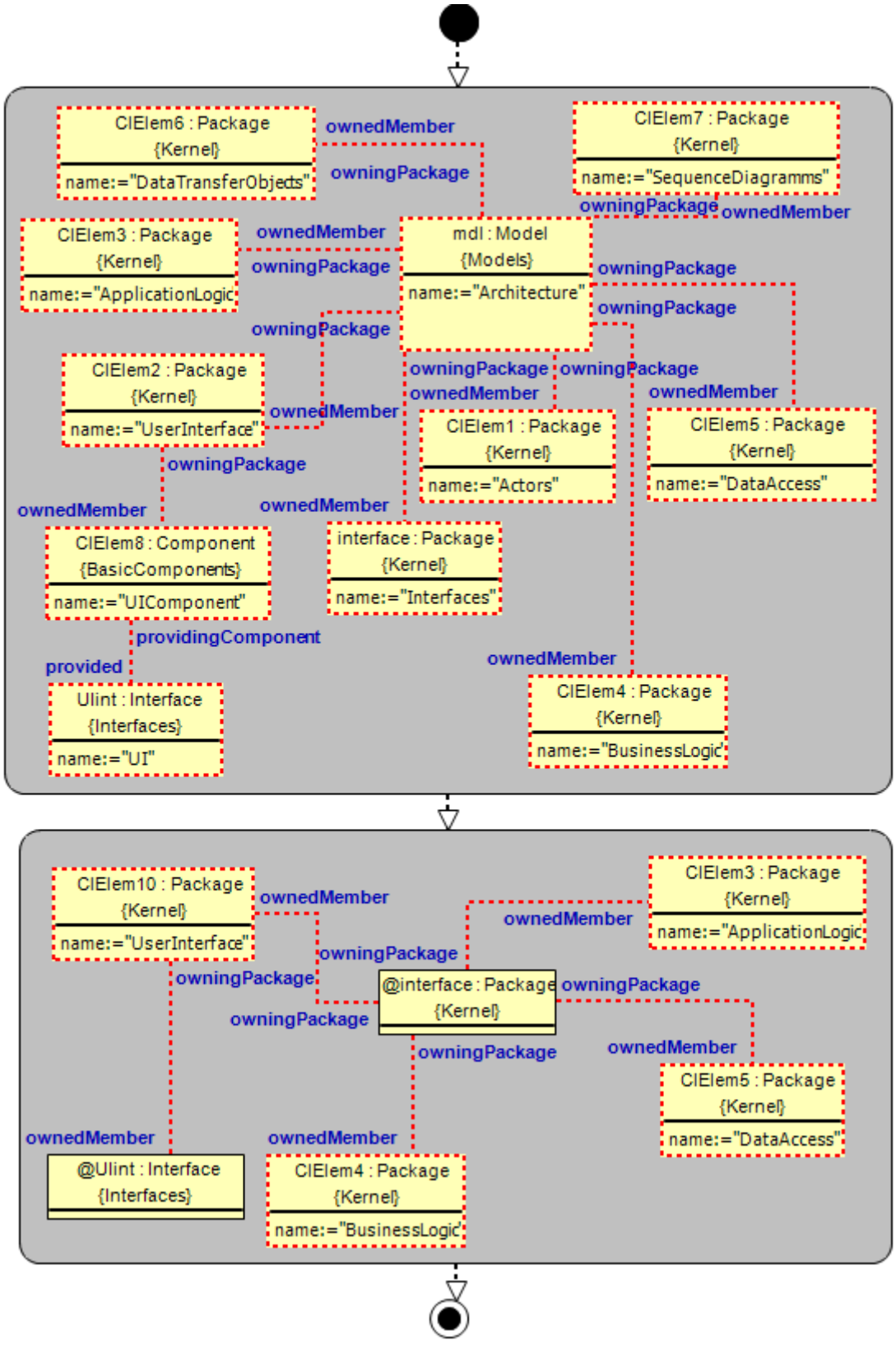

Figure 4.11: Static package creation

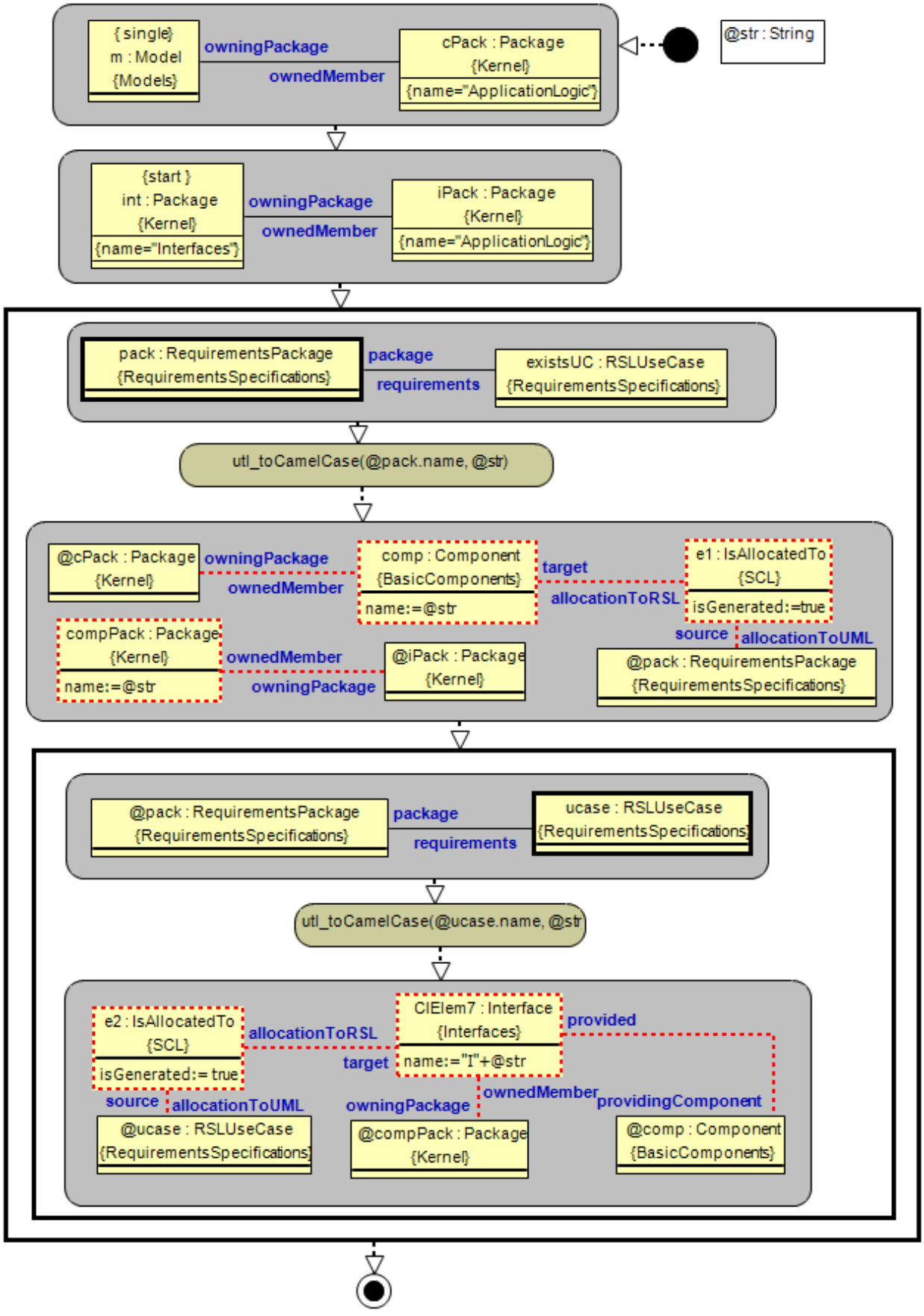

Figure 4.12: Procedure building the Application logic elements

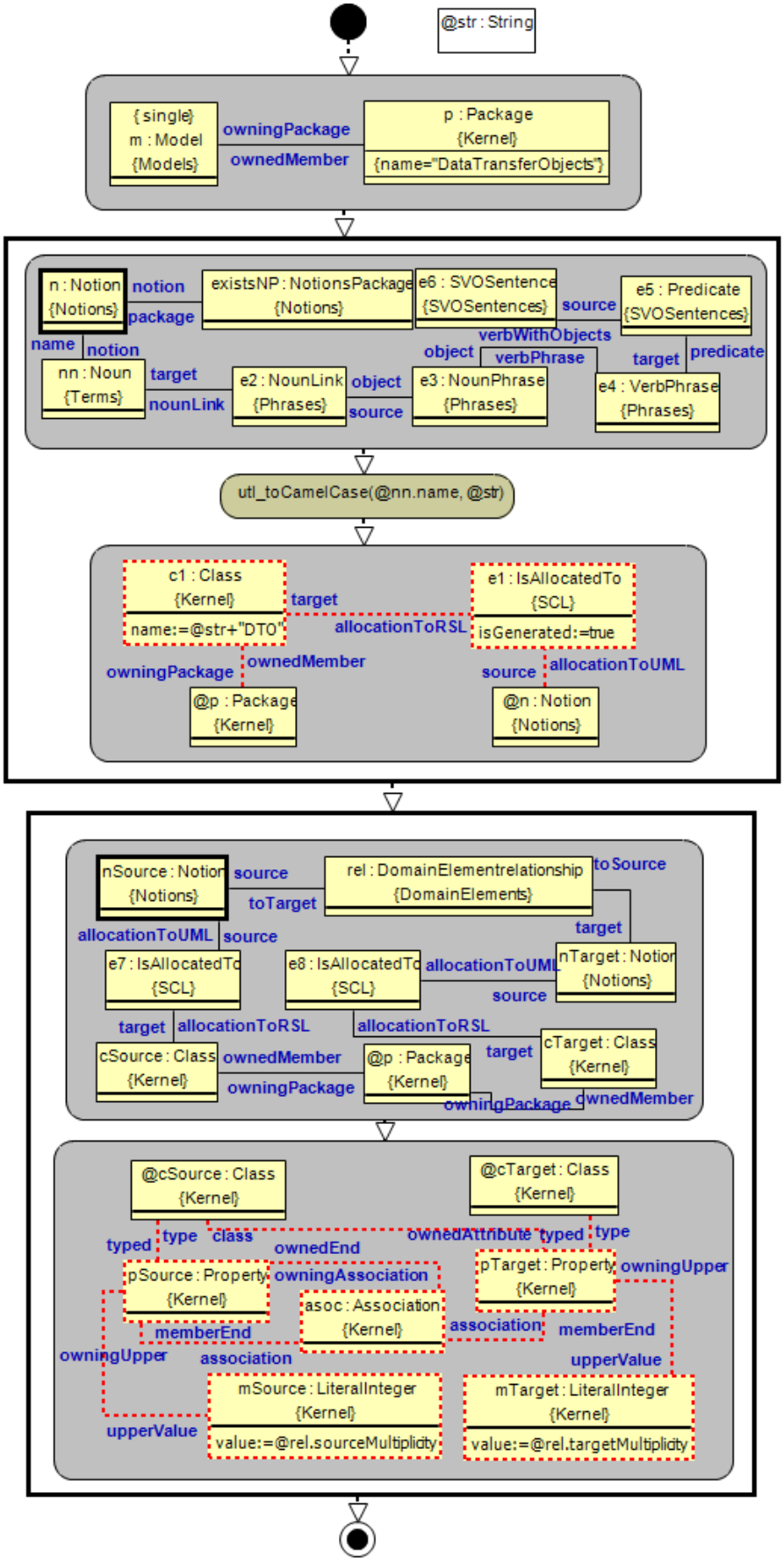

Figure 4.13: Procedure building the data transfer objects

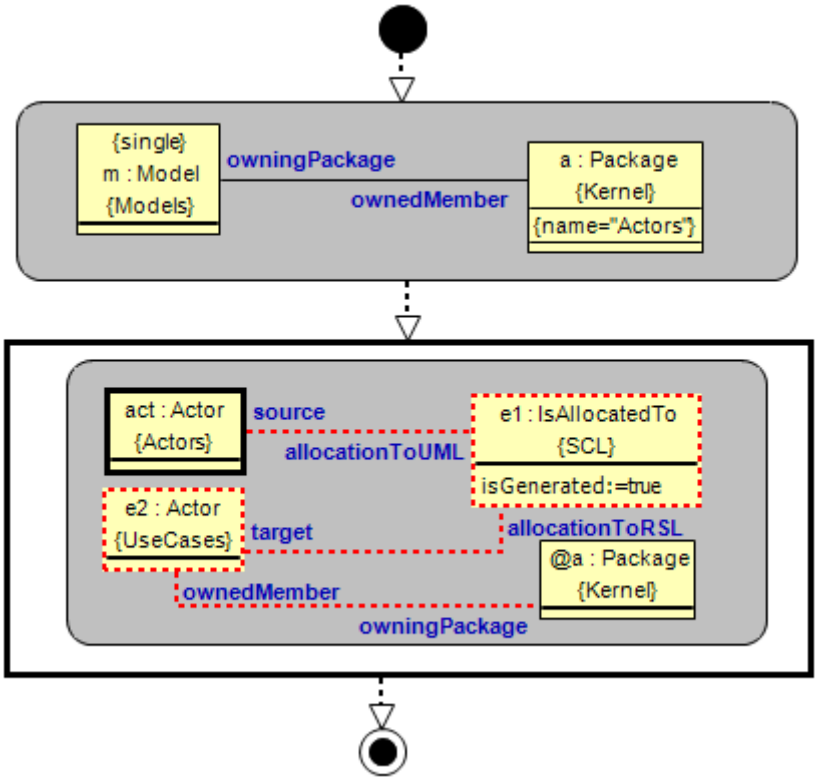

Figure 4.14: Procedure building UML actors

traversal is started, by invoking bhv_ActivityScenarioTraversal 4.23 on the initial node and on the empty interaction.

The recursive procedure bhv_ActivityScenarioTraversal is capable to process one node and add the corresponding messages to the interaction (via bhv_GenerateMessages, required actor lifelines are also created upon request). Then, if only one edge exits the node, the next node is found and the same procedure invoked recursively on this node and the same interaction. If there is a branch node with several edges exiting, a loop is organised, which for every exit creates a new copy of the interaction and invokes the same procedure on this copy and the appropriate node. The tag on the branch is stored, in order to build the correct name of the interaction (currently the name of the last branch is used as a suffix to the use case name). If a final node (or *rejoin* edge) is reached, the current interaction is finalised and this recursive procedure ended. It can easily be seen, that this way the classical graph traversal algorithm is implemented and as many interactions generated, as there are branches in the graph.

In the case of constrained language scenarios, the procedure bhv_CreateTextScenarioInteraction 4.32 initialises an interaction for each available scenario and invokes bhv_CreateTextInteraction-Data 4.33 on this scenario and interaction. This procedure in turn processes language sentences in the defined order and for each does the standard processing via bhv_GenerateMessages. The

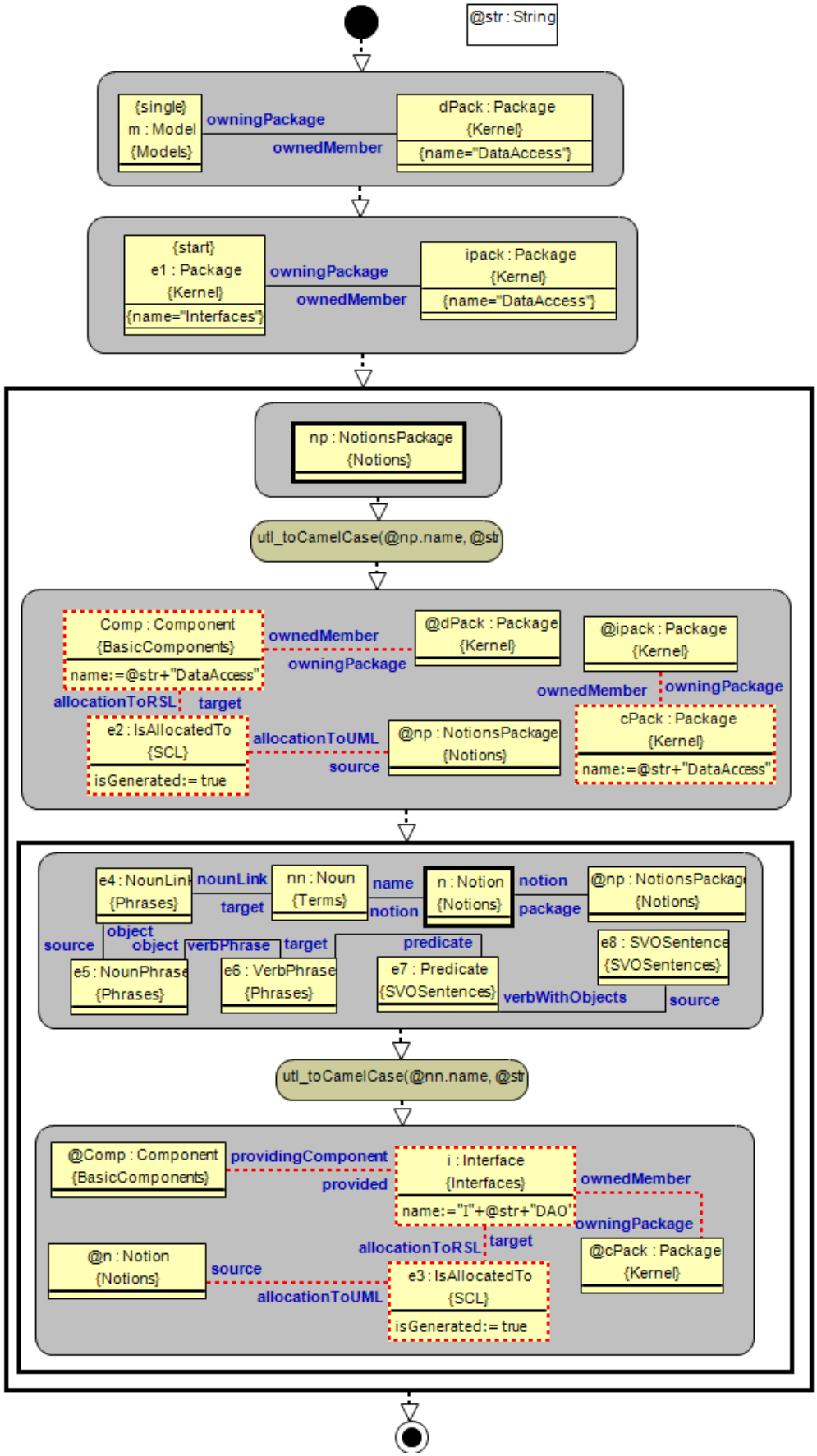

Figure 4.15: Procedure building data access objects

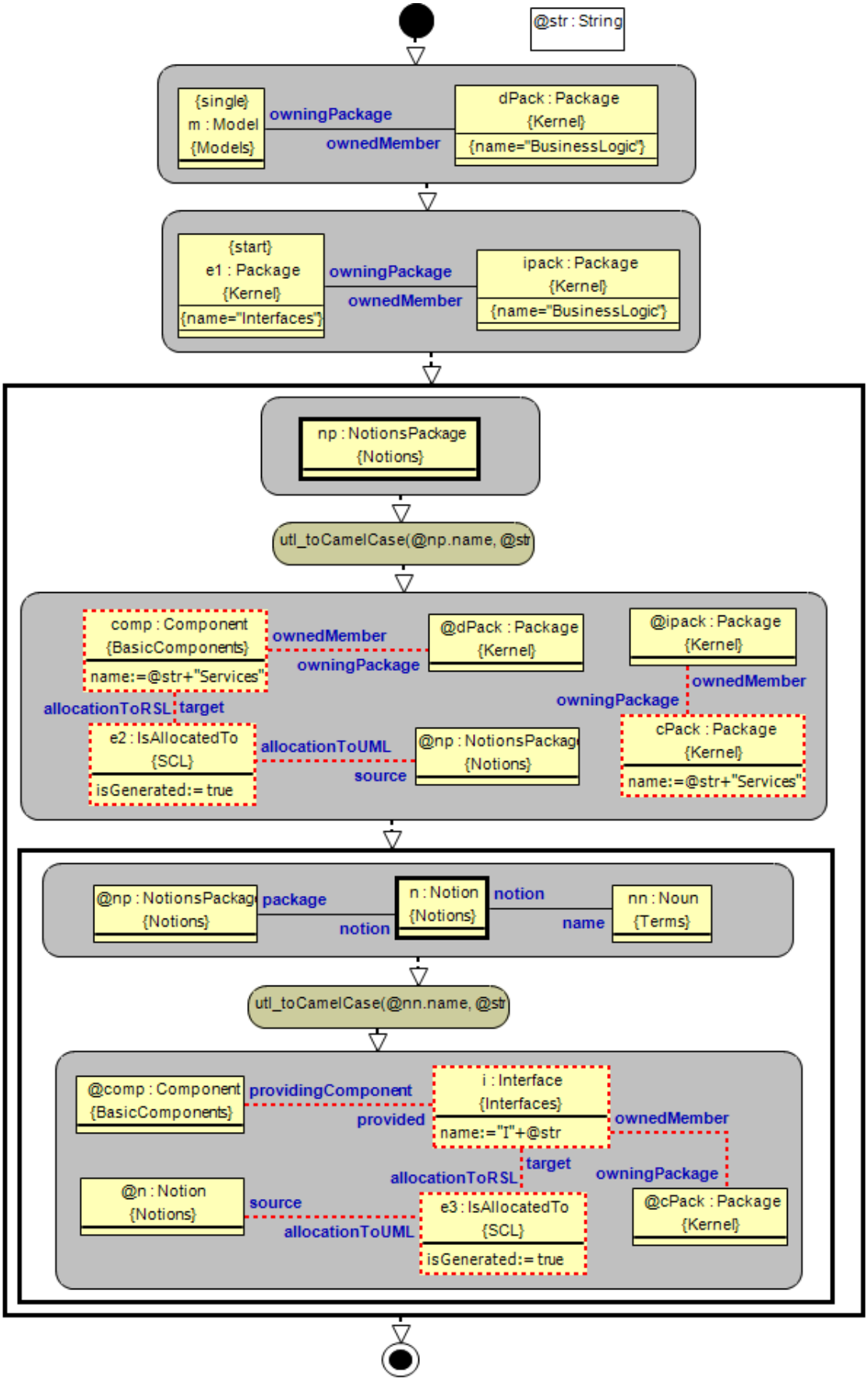

Figure 4.16: Procedure building business logic components

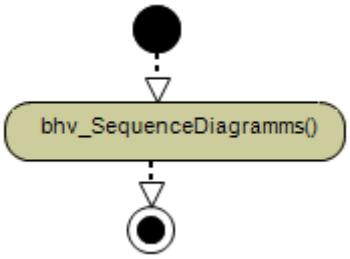

Figure 4.17: Procedure managing the behaviour

only difference from the activity case is that a special construct (using variables) is required to find also the next sentence.

The processing of one sentence by bhv_GenerateMessages (taking into account also the next one) is the same in both cases and similar to the previous version, only all situations are now processed adequately. Required actor lifelines must also be added on demand, because they can differ for several scenarios of one use case. For messages to the system itself the corresponding dependencies must also be built. But otherwise the structure of these procedures is nearly the same as in deliverable 3.2.1 and will not be described in more detail.

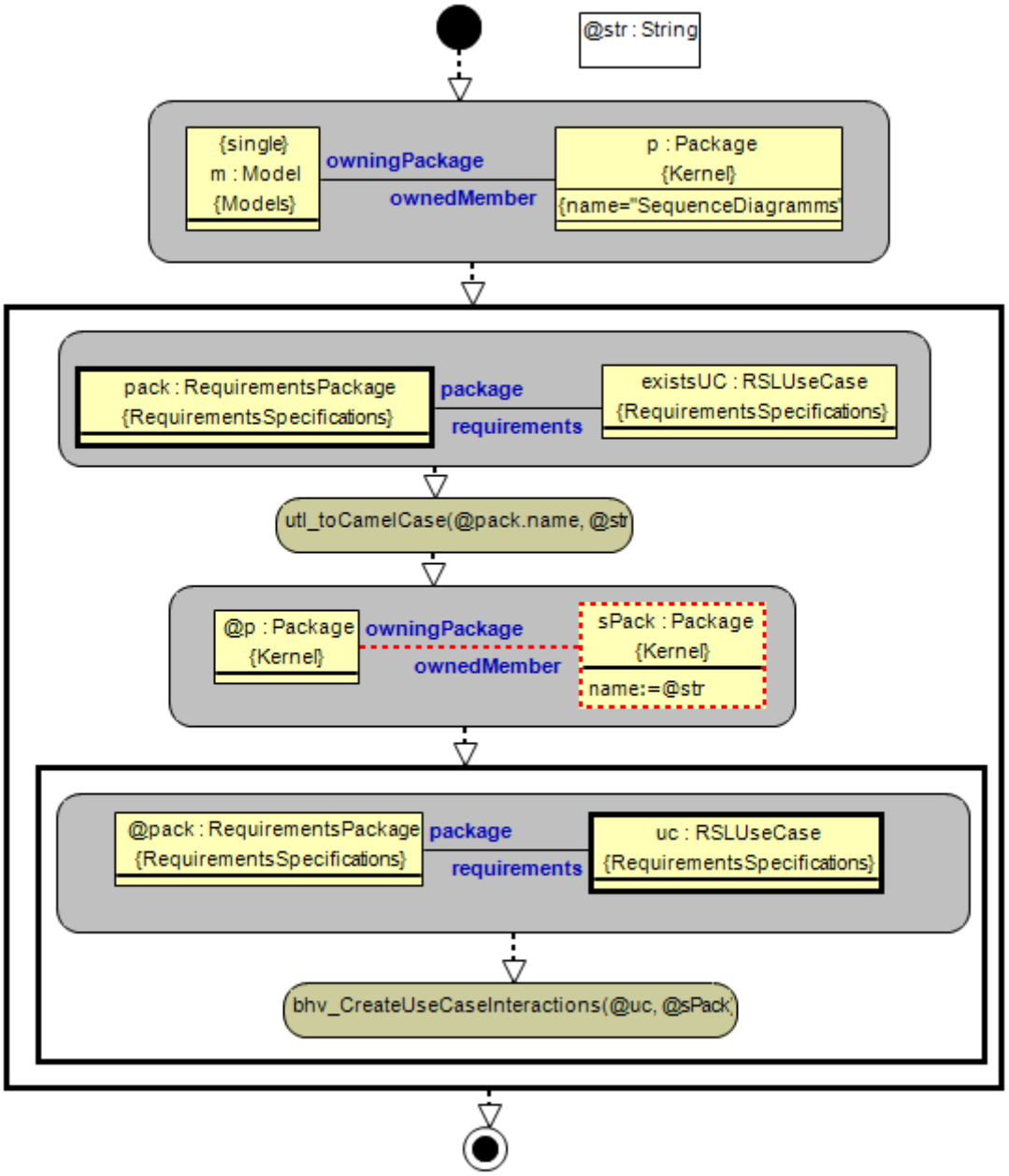

Figure 4.18: Main procedure for behaviour processing

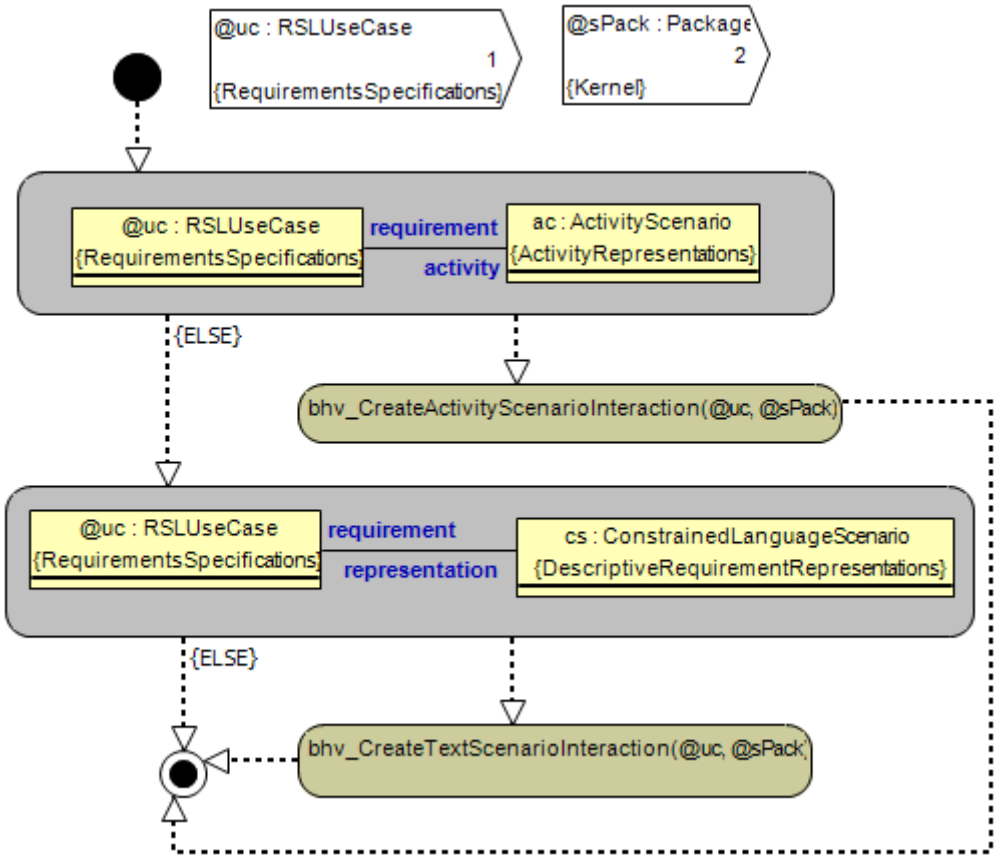

Figure 4.19: Procedure distinguishing scenario kinds

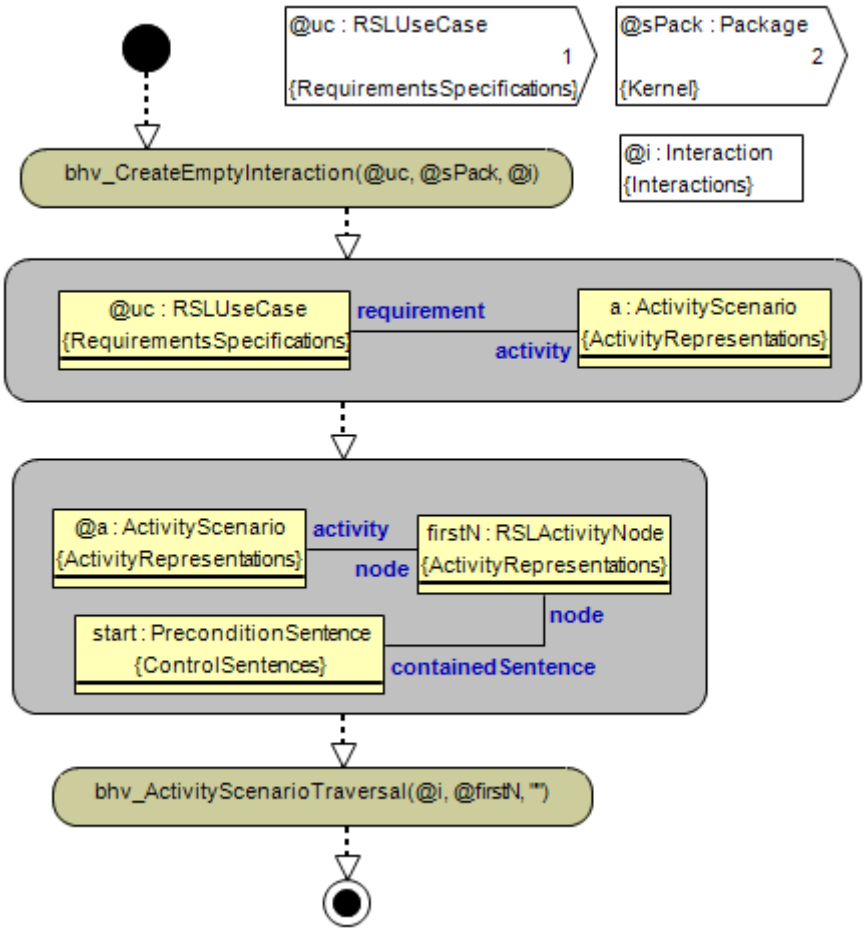

Figure 4.20: Procedure processing activity scenarios

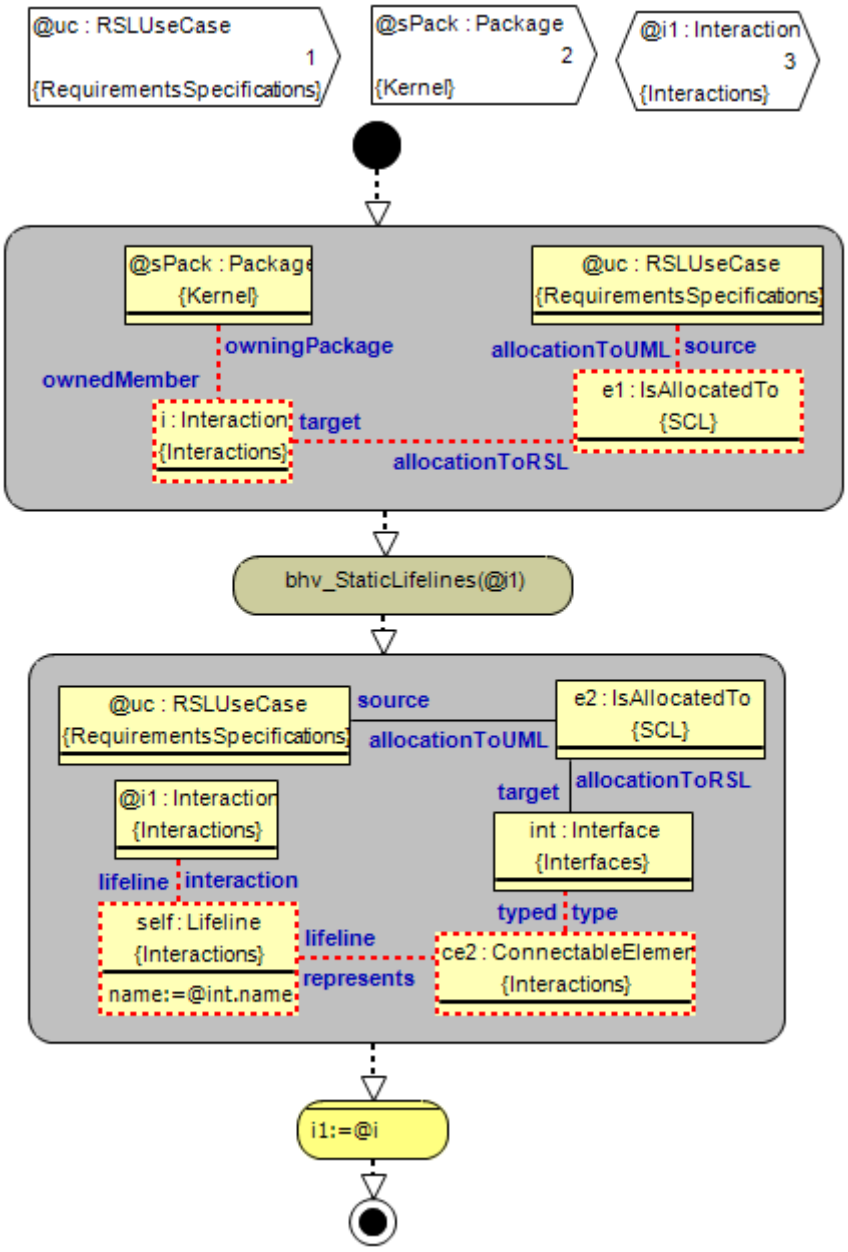

Figure 4.21: Initialiser procedure for activity graph traversing

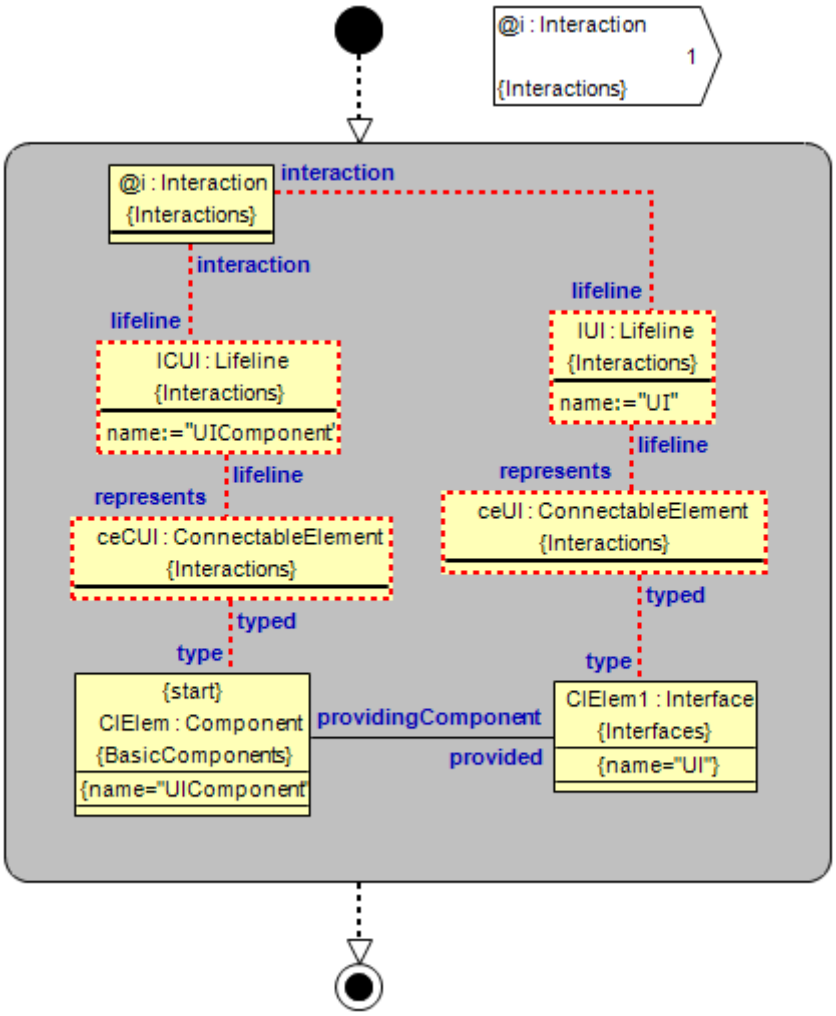

Figure 4.22: Procedure building lifelines

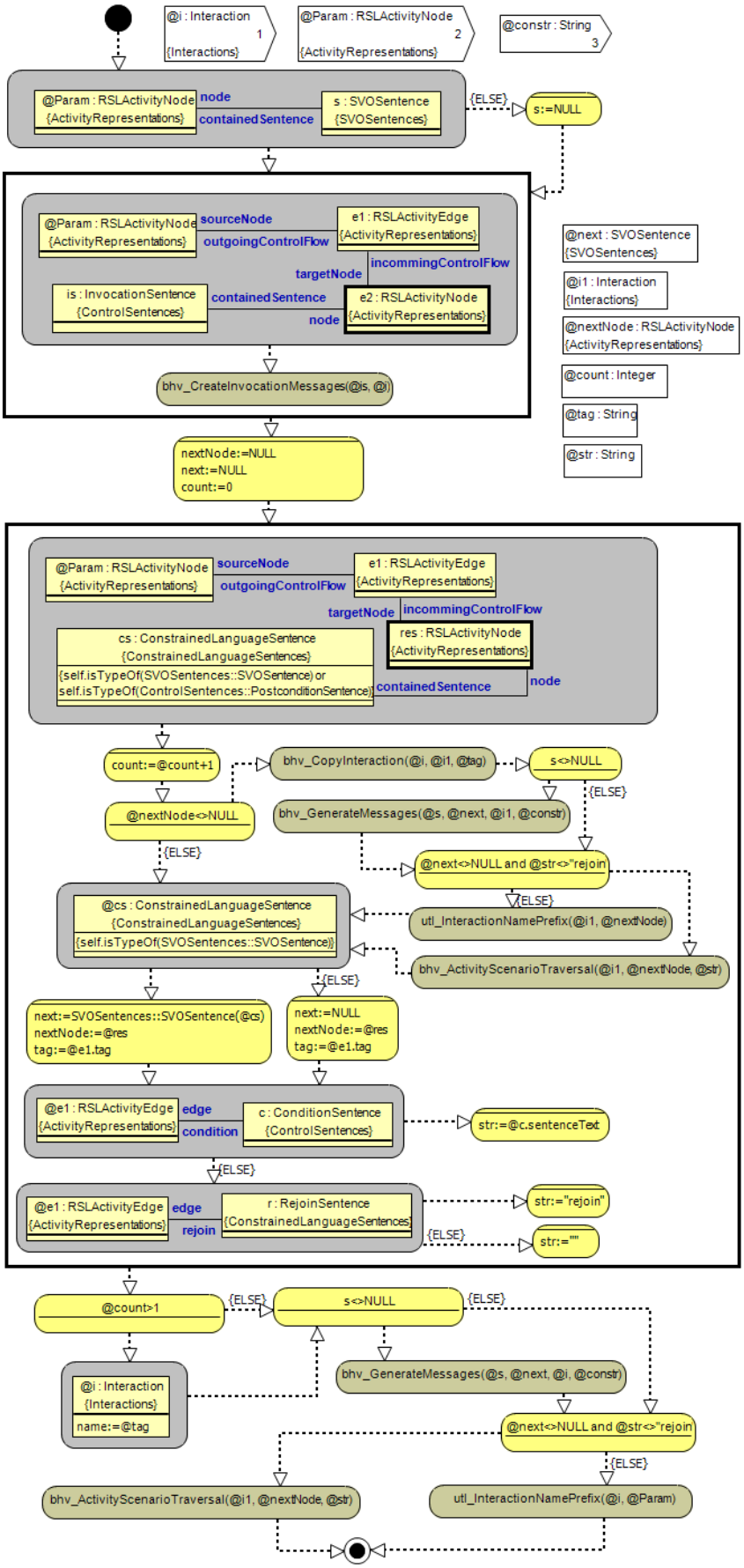

Figure 4.23: Procedure traversing the activity graph

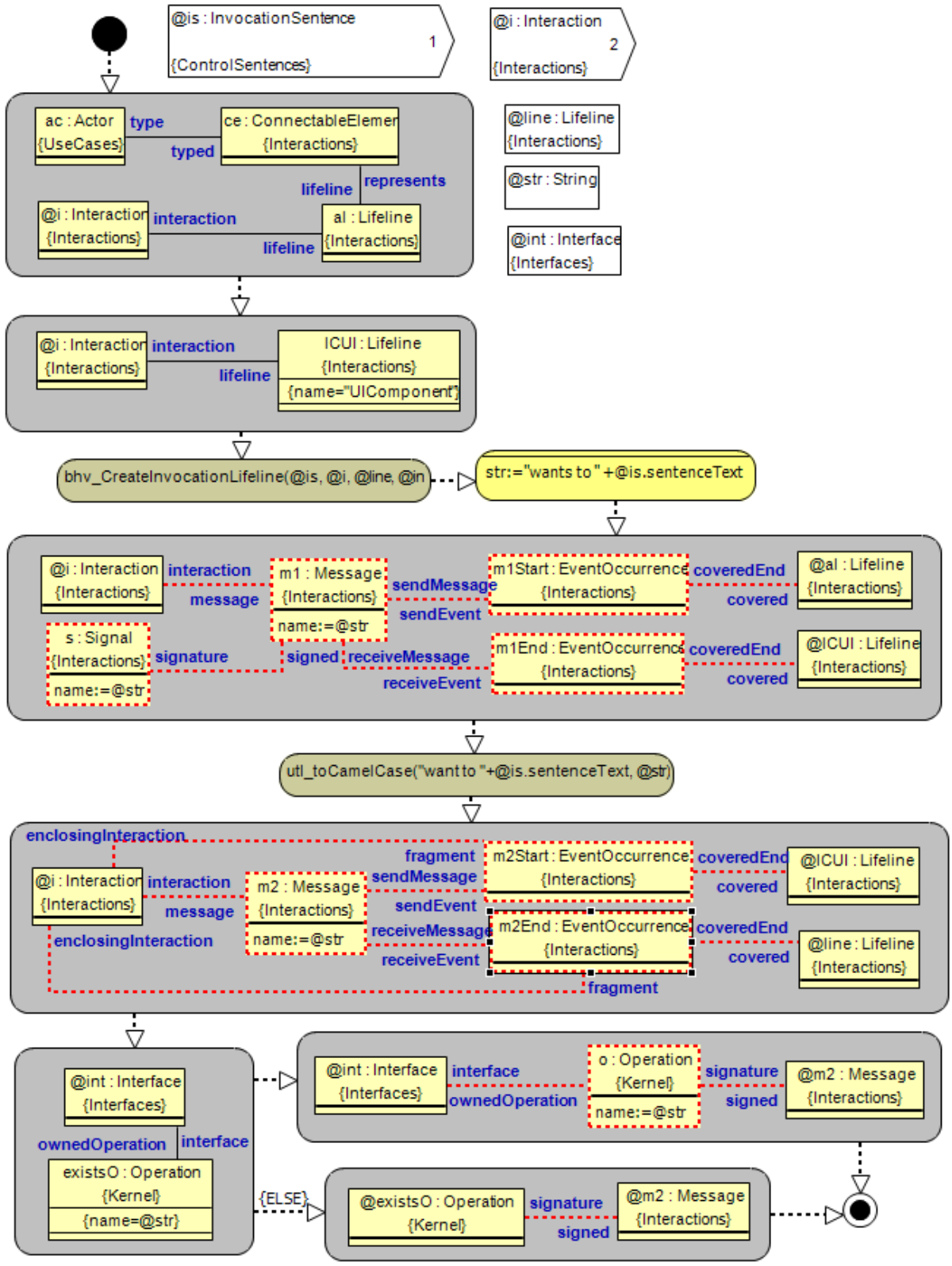

Figure 4.24: Procedure processing invocation messages

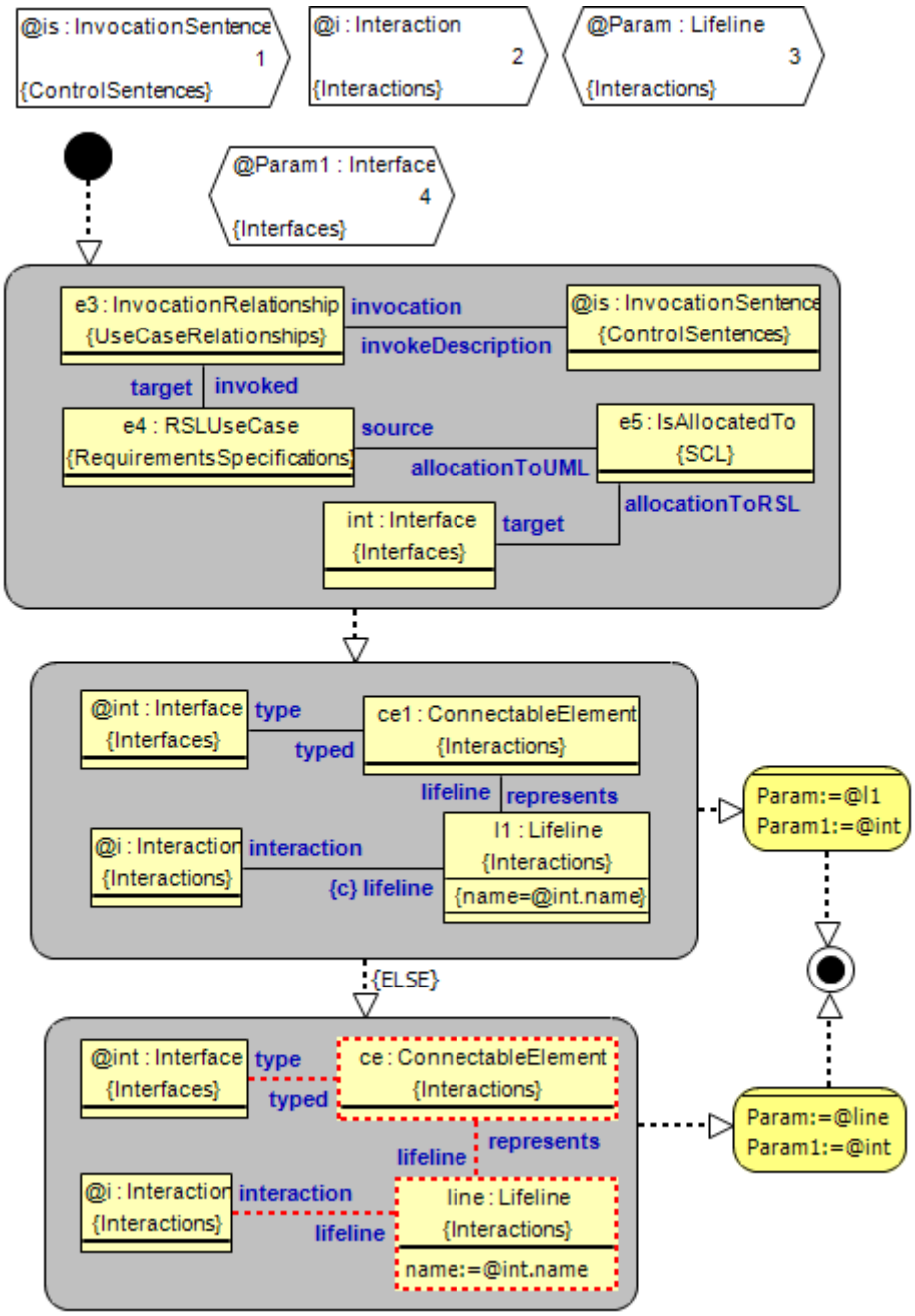

Figure 4.25: Procedure building invocation lifelines

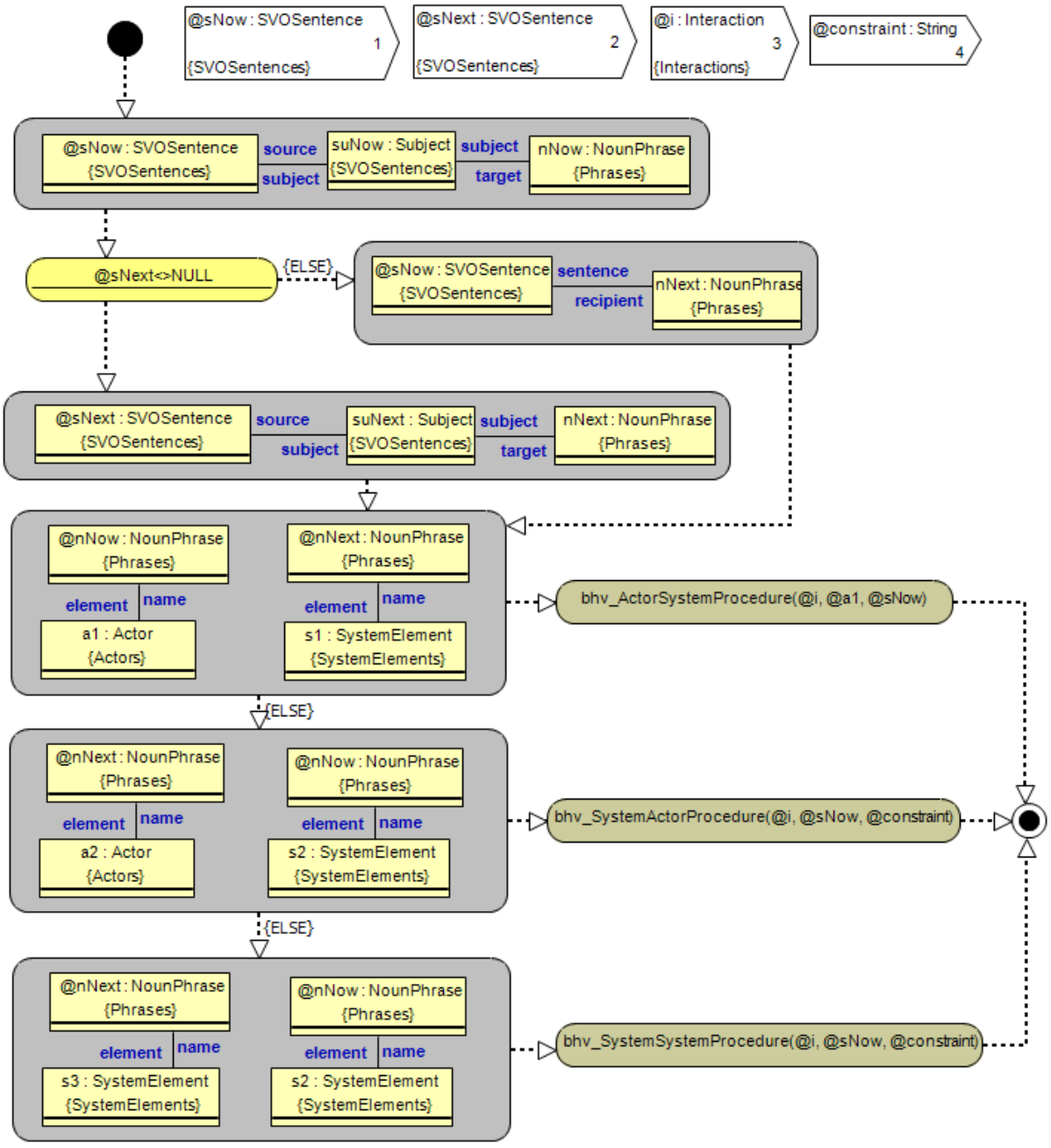

Figure 4.26: Procedure distinguishing message kinds

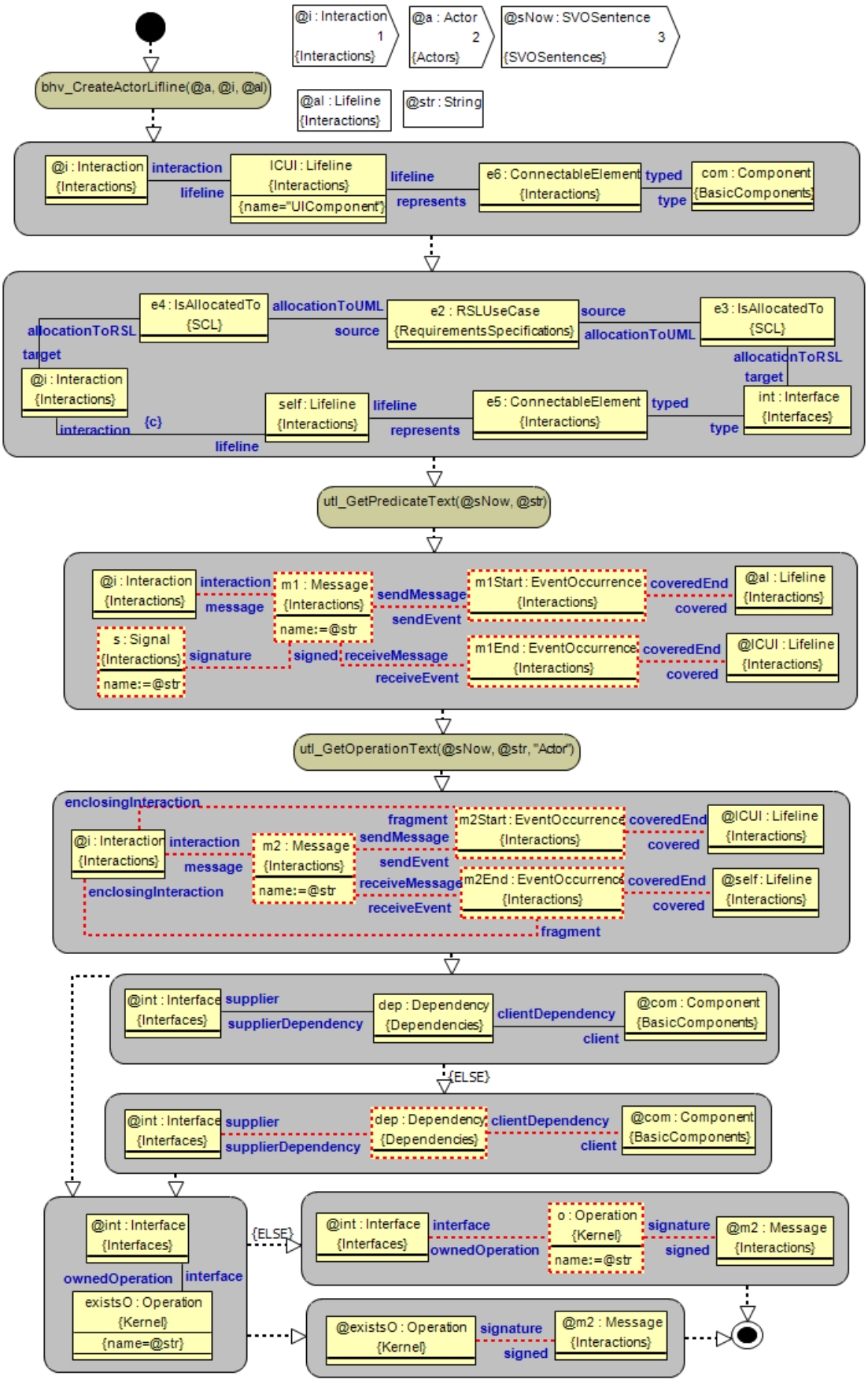

Figure 4.27: Procedure generating actor-to-system messages

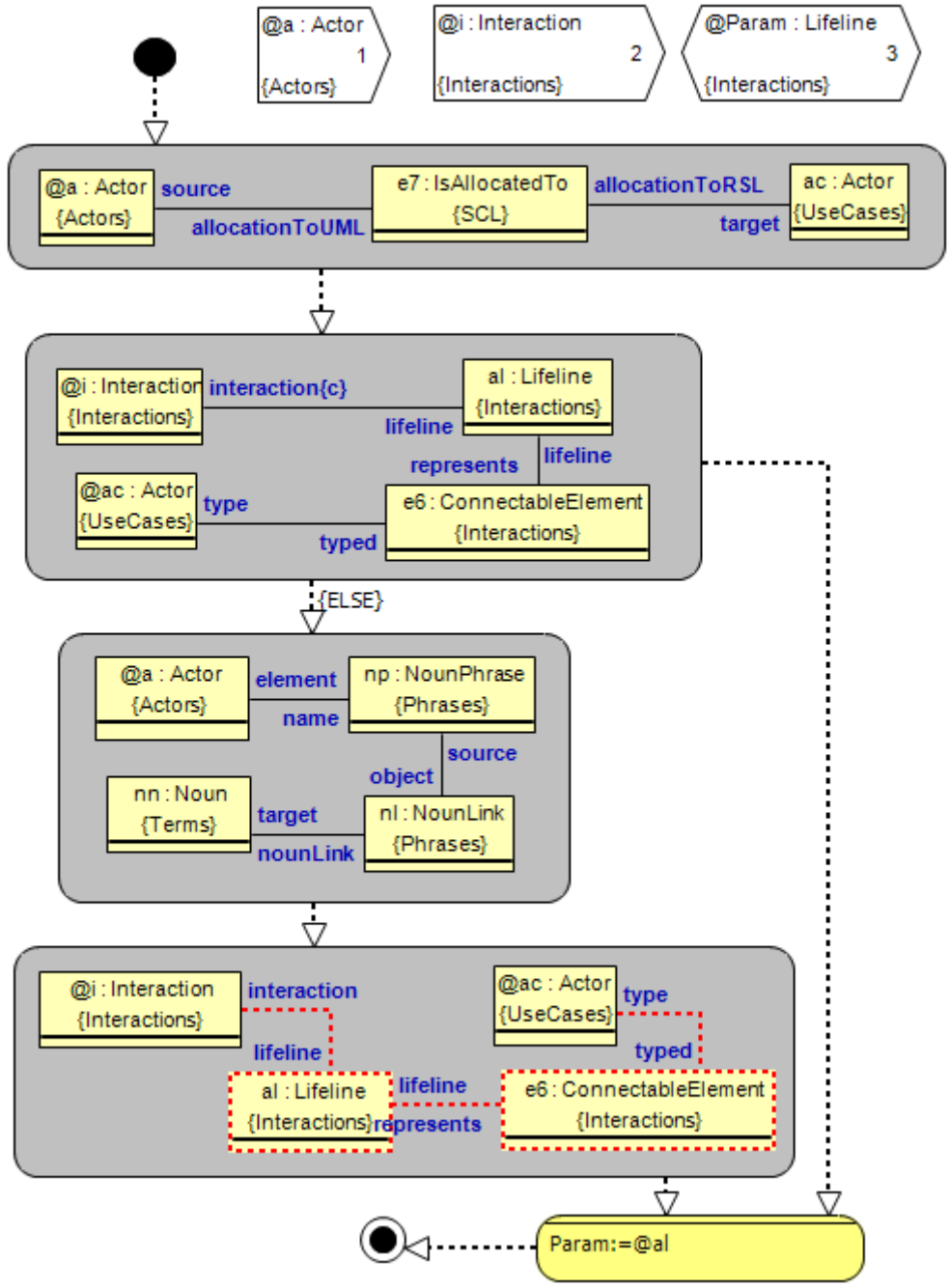

Figure 4.28: Procedure generating lifeline

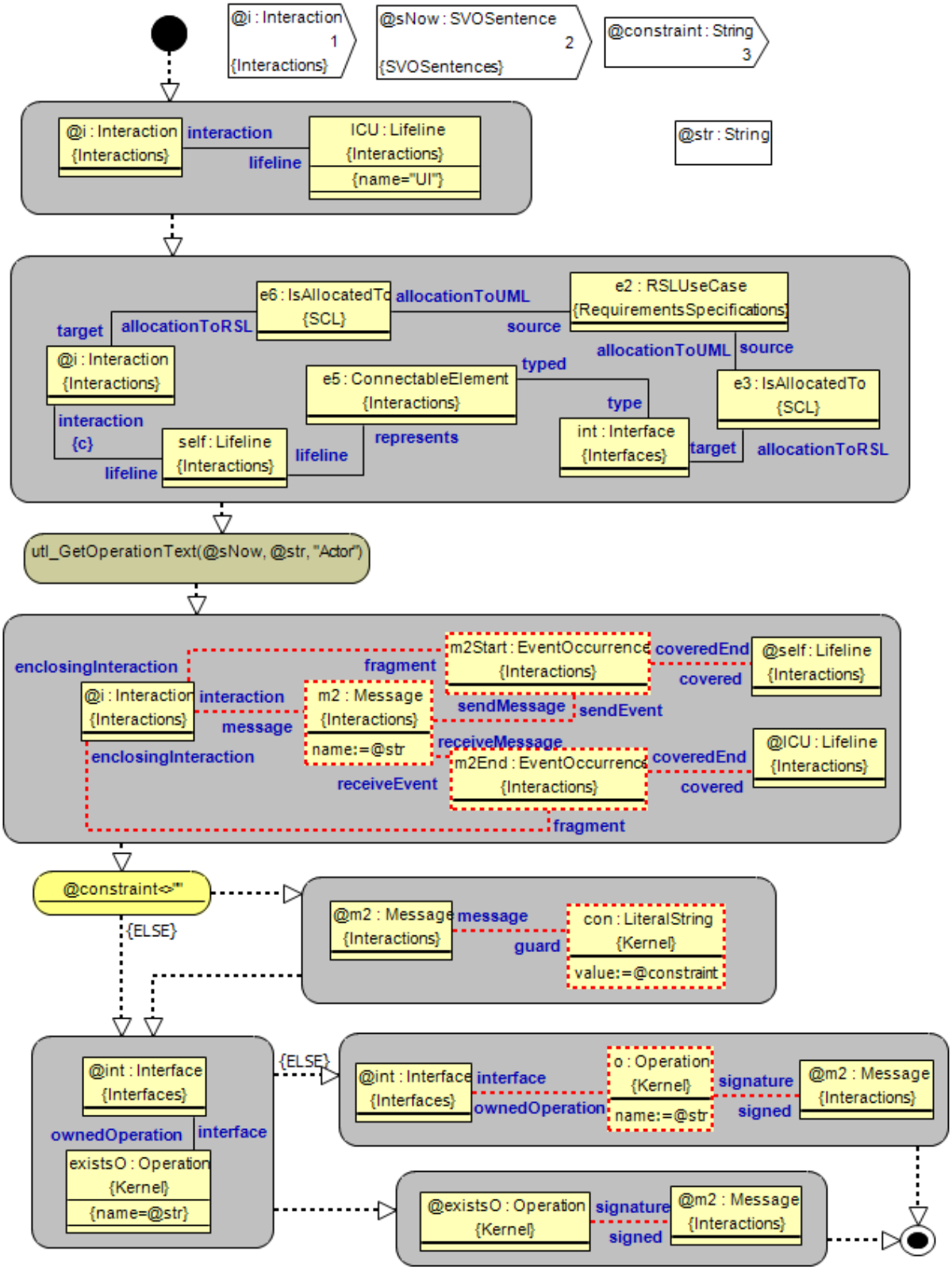

Figure 4.29: Procedure generating system-to-actor messages

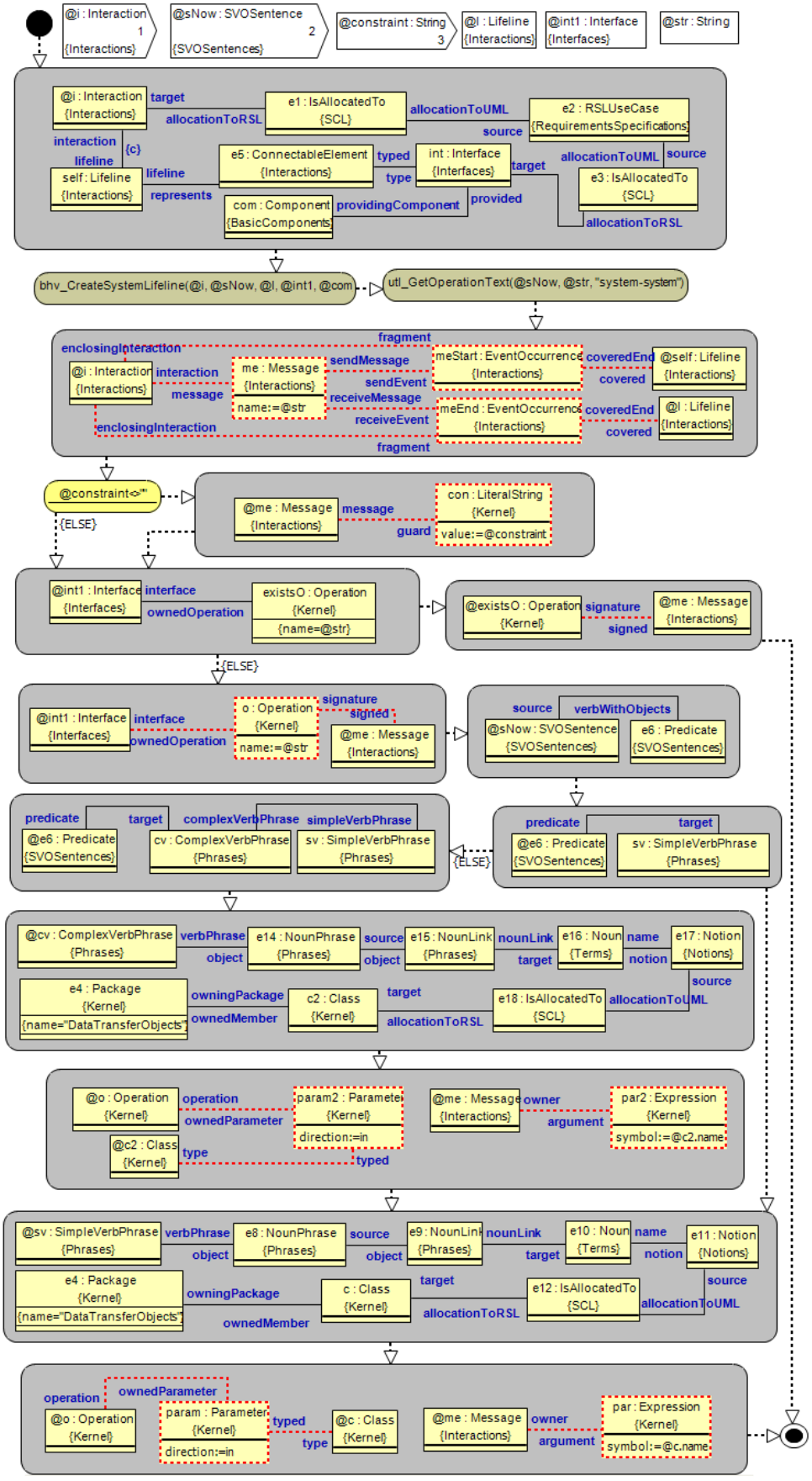

Figure 4.30: Procedure generating system-to-system messages

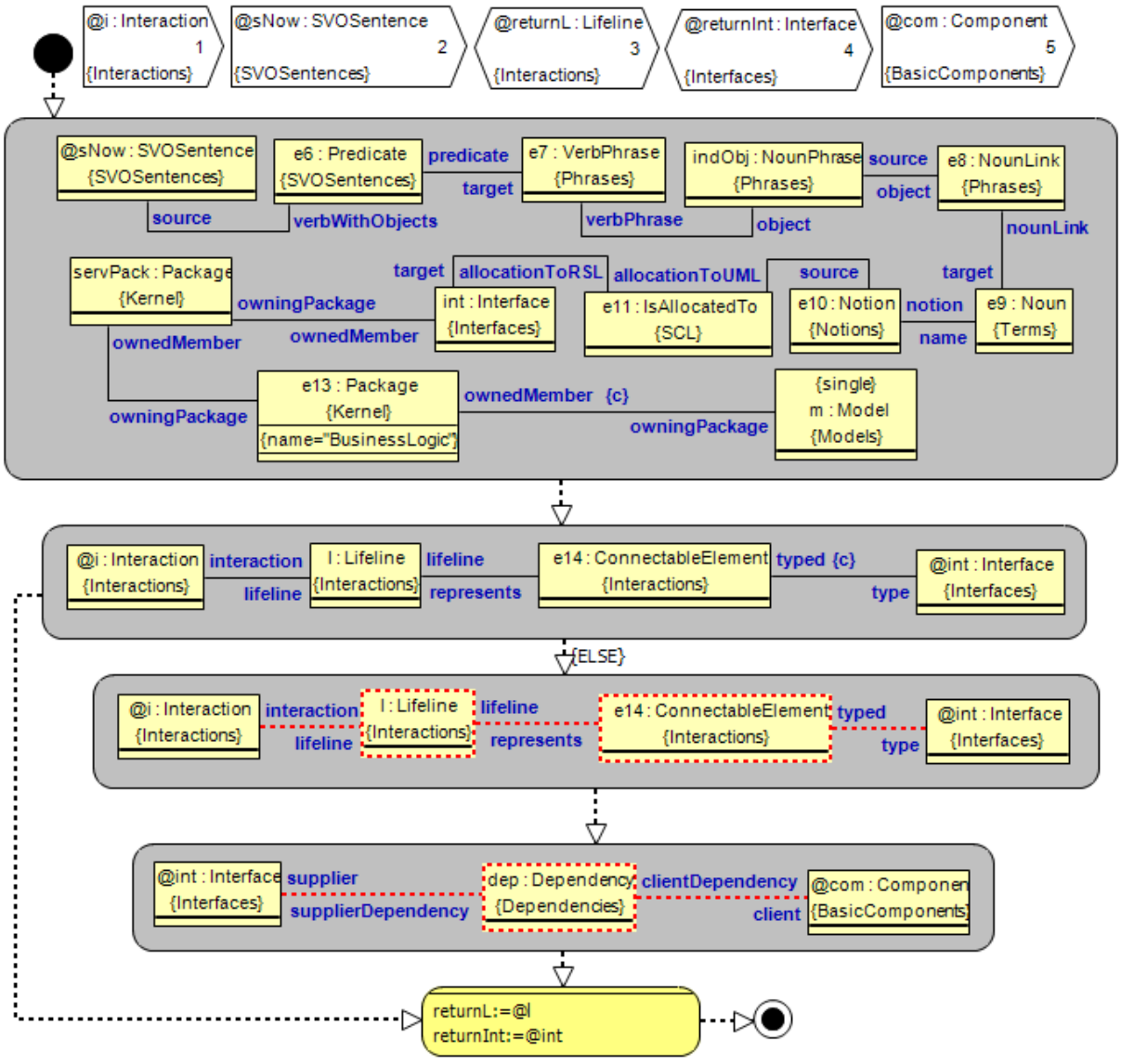

Figure 4.31: Procedure generating system (“business”) lifeline

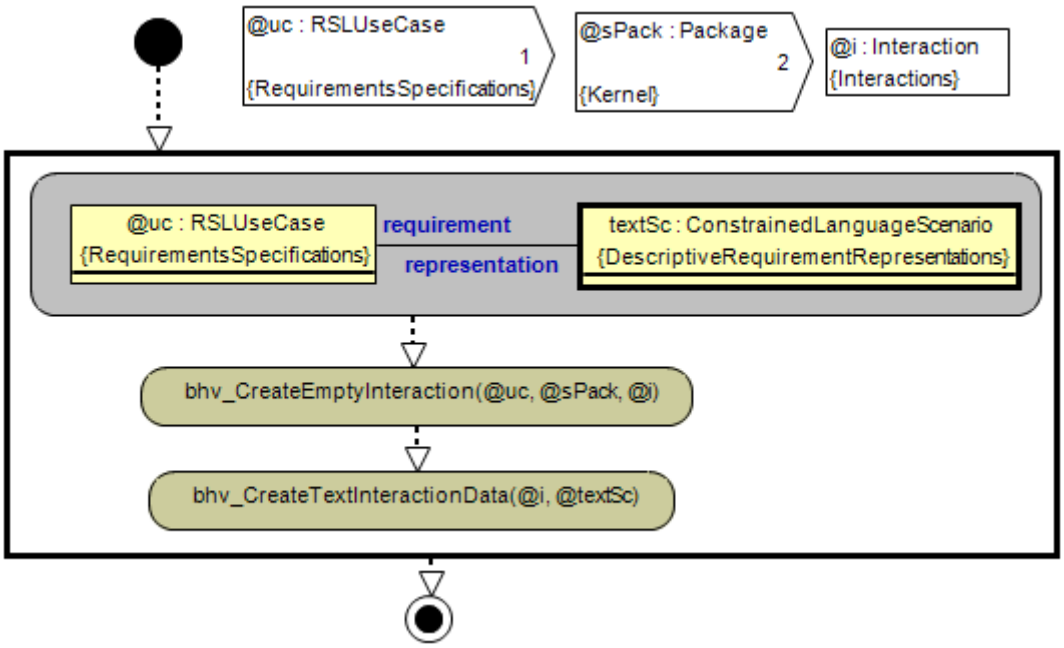

Figure 4.32: Procedure initialising interaction for constrained language scenario

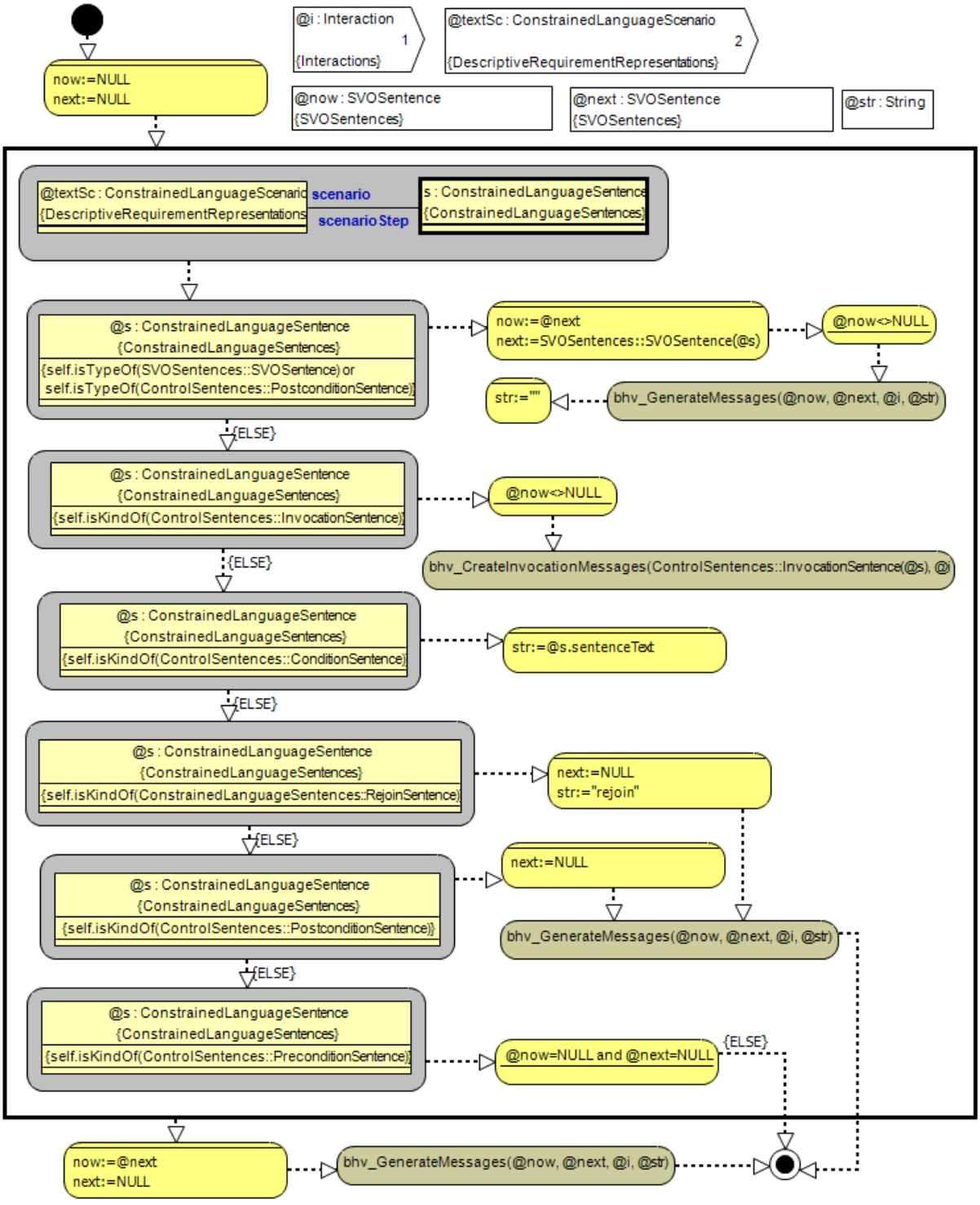

Figure 4.33: Procedure processing constrained language scenarios

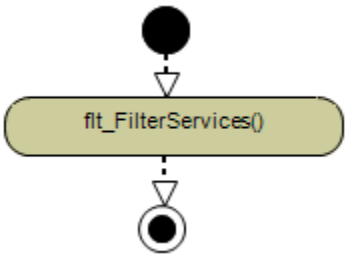

Figure 4.34: Main procedure for behaviour processing

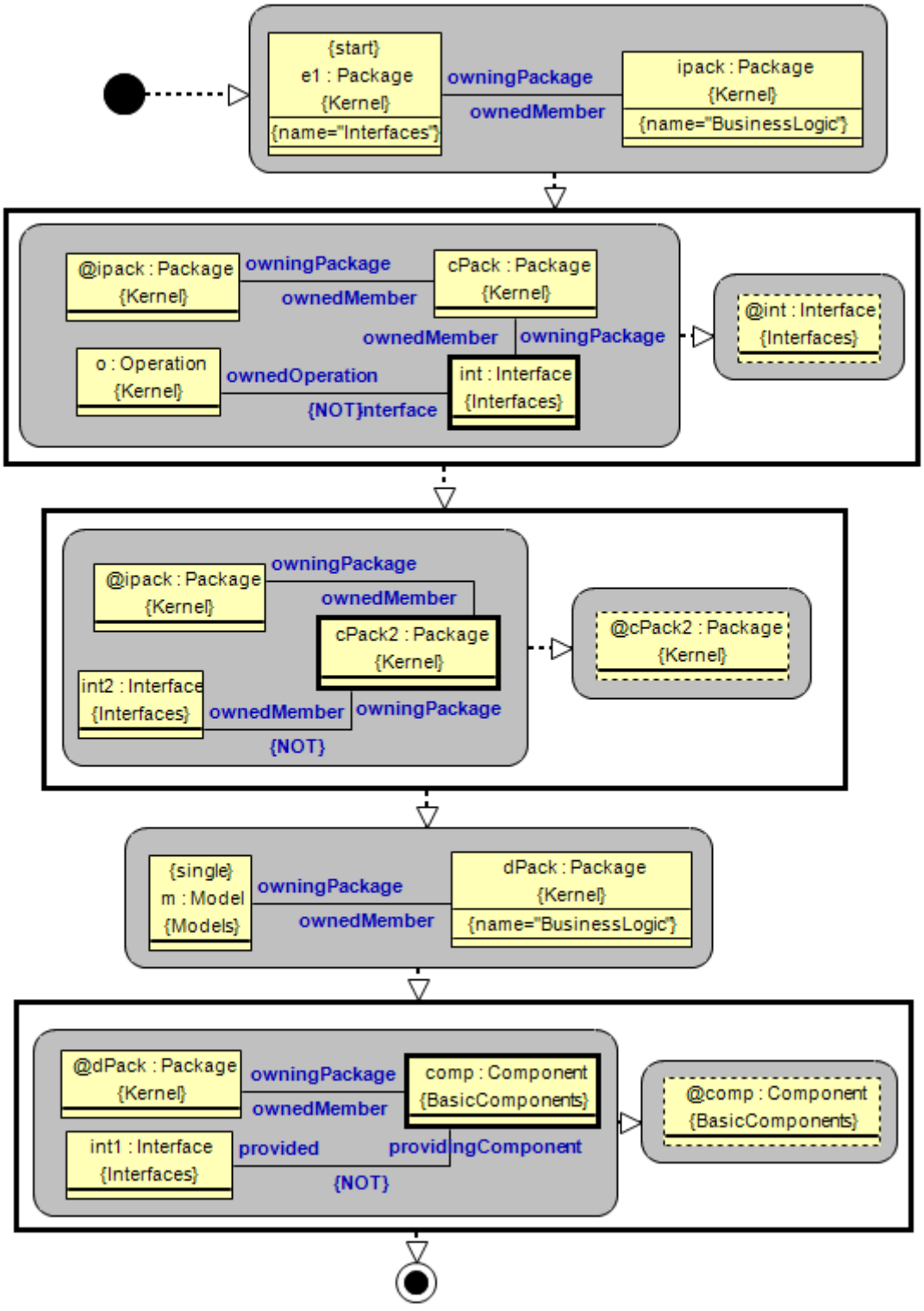

Figure 4.35: Procedure for deletion of unused interfaces

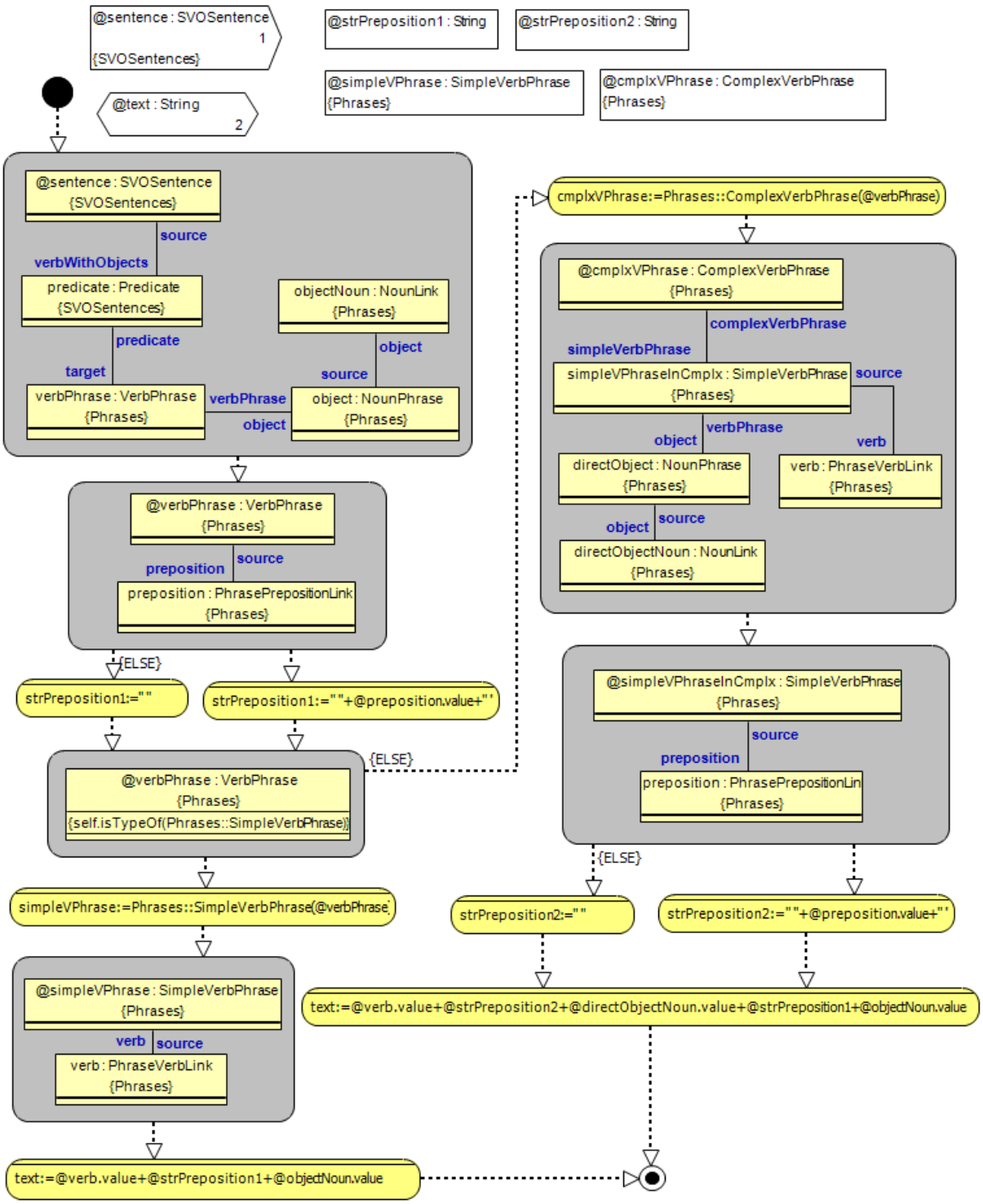

Figure 4.36: Utility procedure providing verb phrase text

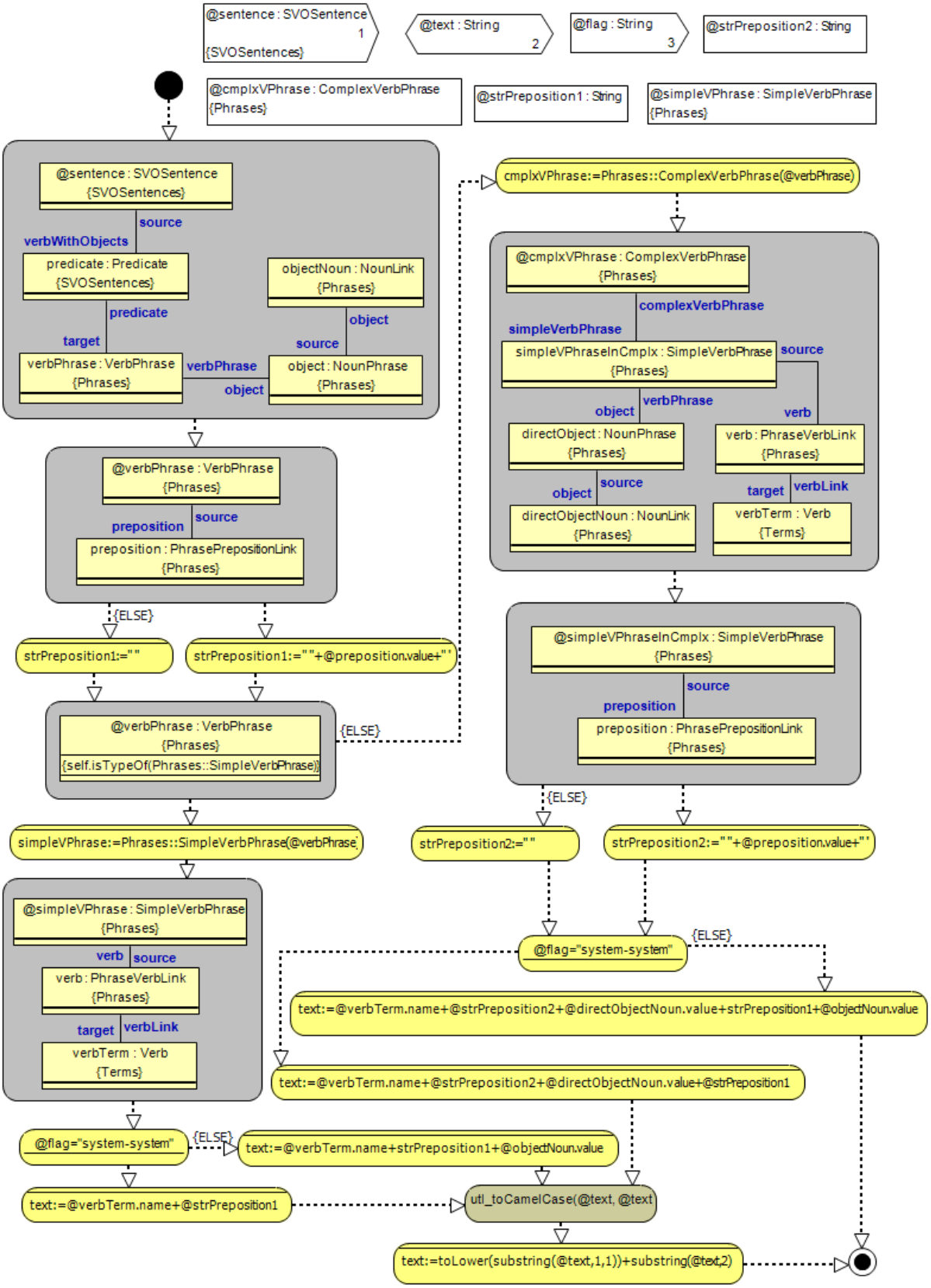

Figure 4.37: Utility procedure providing operation name

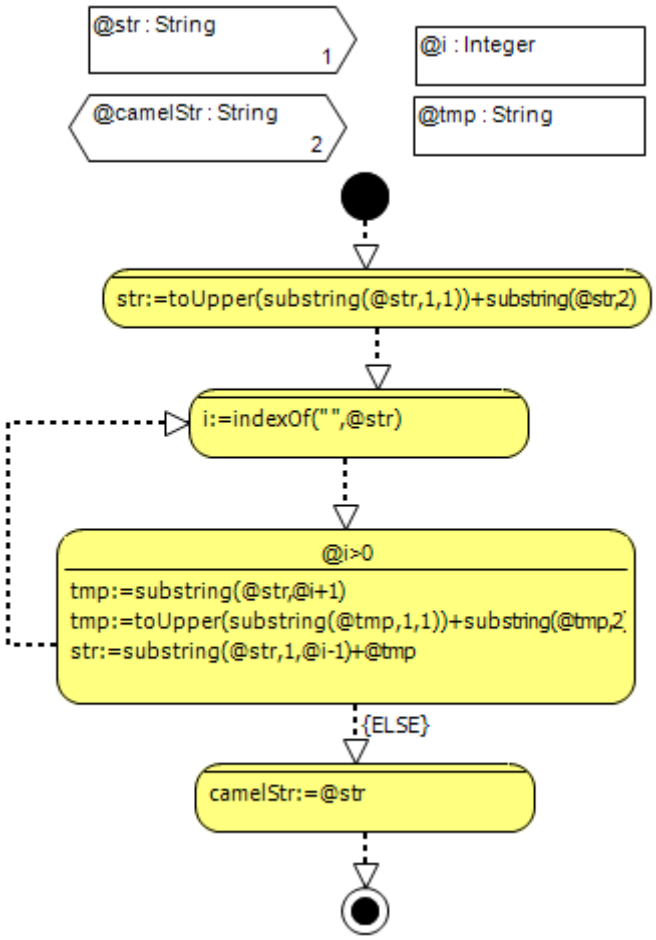

Figure 4.38: Utility procedure converting string to UpperCamelCase

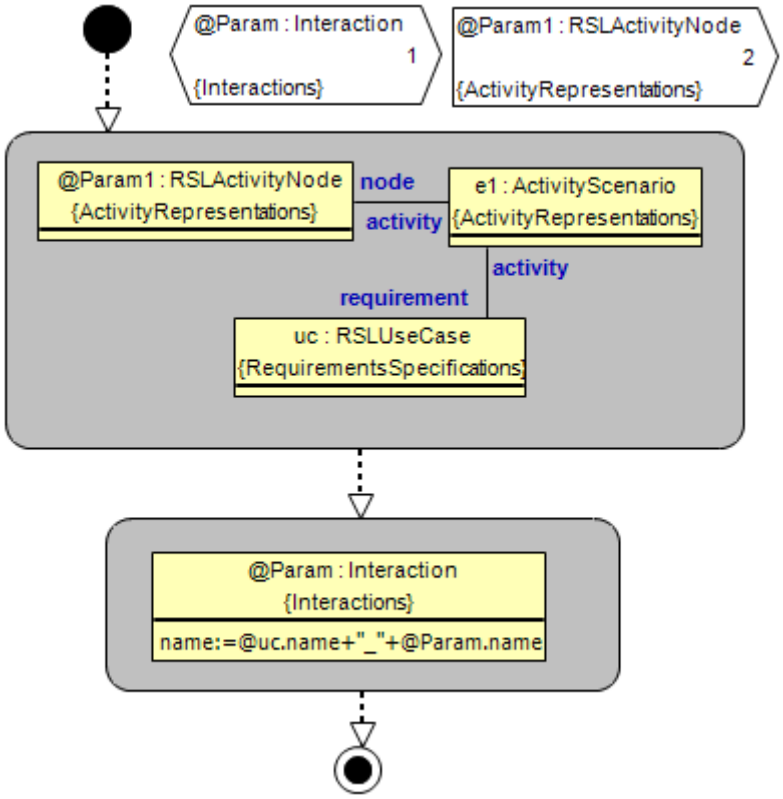

Figure 4.39: Utility procedure providing interaction name prefix

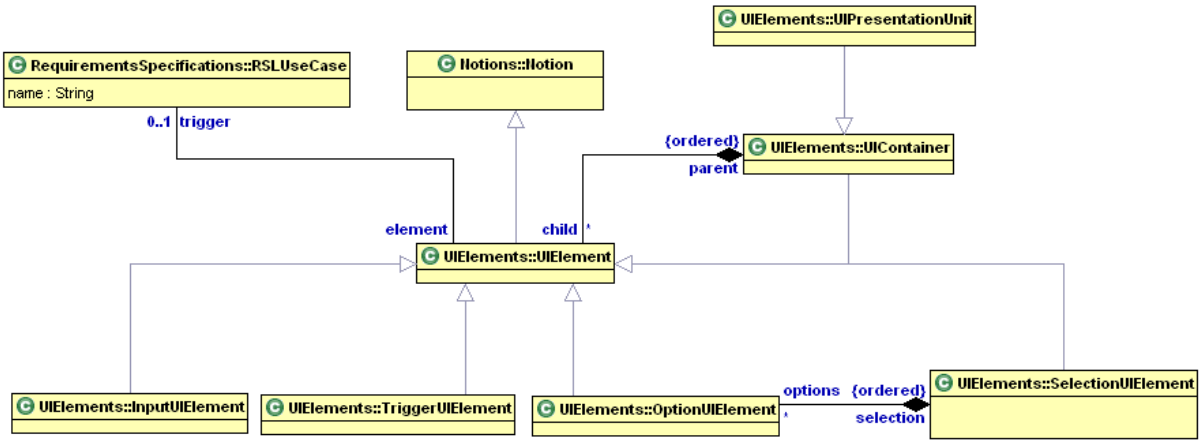

Figure 4.40: UI elements and relationships

4.3 Transformations of UI elements in RSL

This section formalises the transformation of UI elements defined in section 3.3. MOLA transformations require metamodels restricted basically to EMOF as already mentioned in section 4.1. To be able to transform user interface requirements, a MOLA-ready version of the RSL UI metamodel part and the GUI metamodel is required. Based on these MOLA-ready source and target metamodels, additional MOLA transformation rules for the UI specification are defined. The remainder of this section is organised as follows: First, the user interface specific additions to the MOLA-ready version of RSL are explained, Second, the target GUI metamodel is described. Third, MOLA rules which formalise the rules of section 3.3 are specified and in the last step, MOLA rules that extend the behaviour of the UI component are defined.

4.3.1 Source and target metamodels for UI transformations

Figures 4.40 and 4.41 illustrate the MOLA-ready version of the user interface part. Since, the UI part of RSL does not rely on CMOF constructs, the RSL UI elements and RSL UI behaviour representation packages have been added to the MOLA-ready RSL version without modifications. Only missing association role names had to be added to make them accessible in the MOLA tool.

For the target metamodel we stick to the MOLA-ready UML version. The UML profile mechanism is used to describe target GUI models conforming to the GUI metamodel. In detail, only UML stereotypes are used to specify instances of the GUI metamodel. It has to be noted that in most projects and GUI frameworks the model-to-code transformation generates only one class for each dialog, containing initialisation code building up the GUI structure according to the

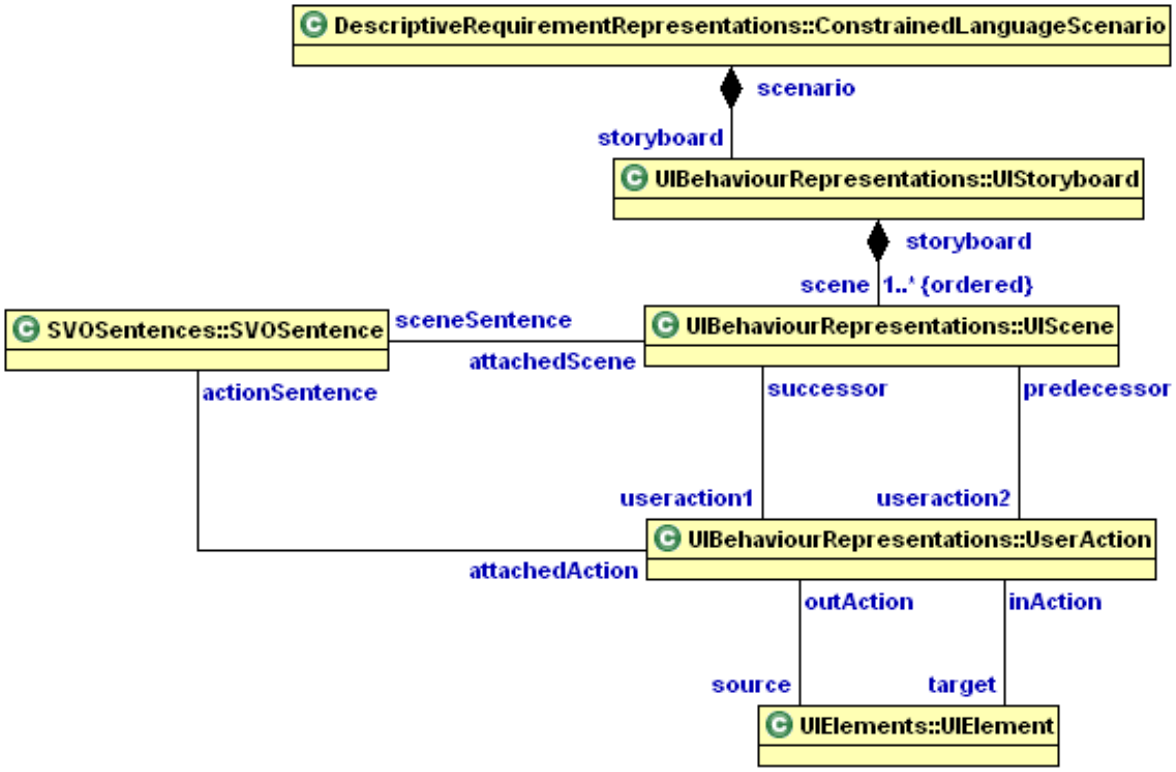

Figure 4.41: UI behaviour representations

GUI model. Hence, in contrast to the other parts of the design model, generated UML classes with stereotypes conforming to the GUI metamodel except “Dialog” and “Frame” will be not transformed into code classes.

4.3.2 Generating dialog structures

The following rules specify the generation of dialogs and their structure. The main procedure is illustrated in Figure 4.42. It requires a UML package as input to which the generated classes are added. This package is generated by rules generating the 4-layer architecture. The rules described here already present a design for the UI component. Thus, they can be also executed during detailed design transformation specified in the next section.

The procedure shown in Figure 4.42 generates a dialog for each UI presentation unit and a traceability link (isAllocatedTo) between the UI presentation unit and the generated dialog. Furthermore, the procedure adds recursively panels to the dialog that correspond to the UI container hierarchy and makes the panels traceable to their UI container.

After generating the primary dialog structure, the procedure in Figure 4.43 processes all other UI elements except UI containers. It distinguishes the different UI elements and calls type-

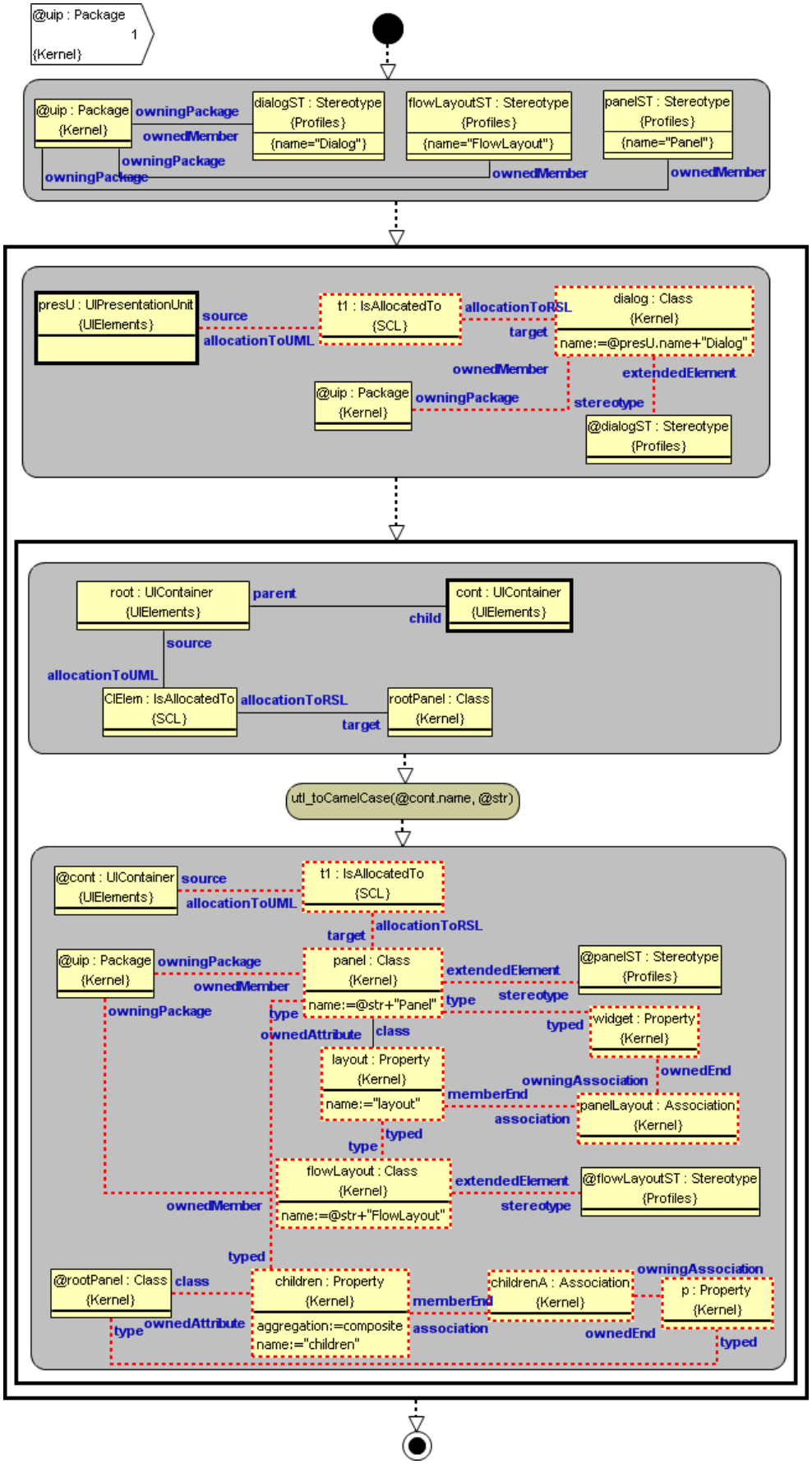

Figure 4.42: Procedure generating GUI dialogs together with traceability links

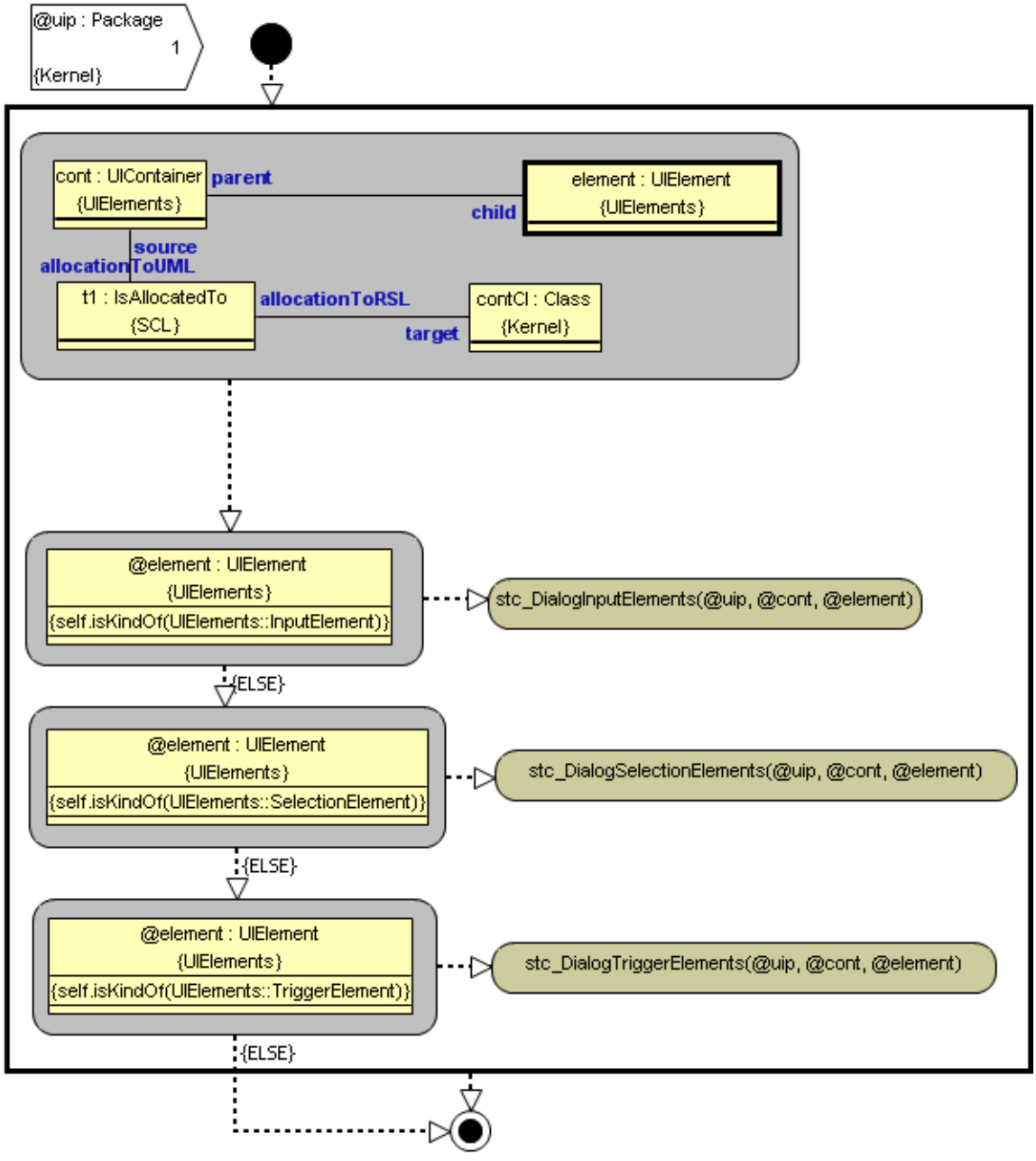

Figure 4.43: Decision between different UI elements

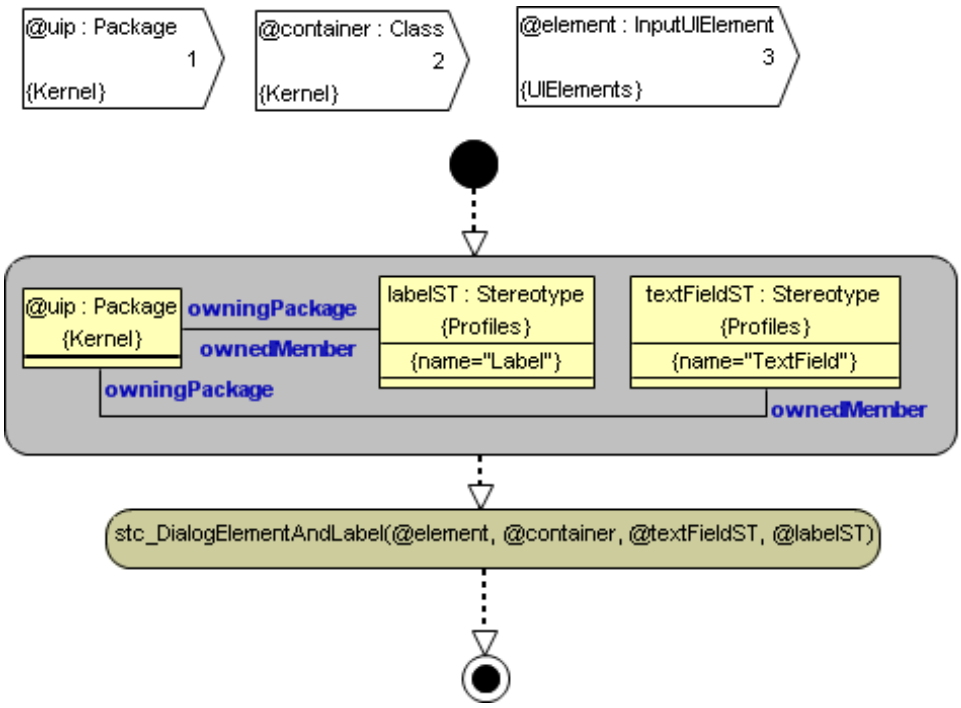

Figure 4.44: Procedure generating GUI elements for input UI elements

specific subprocedure and passes the element, its container and the UML package to the sub procedure.

The procedure shown in Figure 4.44 is responsible for generating text fields out of input UI elements. It simply calls the procedure responsible for generating the concrete widget with an associated label. The procedure in Figure 4.44 selects the appropriate stereotypes and hands them over to the subprocedure which then is able to generate correctly stereotyped UML classes.

The procedure shown in Figure 4.45 is similar to the prior procedure but responsible for selection UI elements. The threshold for distinguishing between radio buttons or check boxes and list boxes on the other hand should be adapted to the needs in each project according to usability guidelines. This procedure distinguishes between four cases:

- Only one option is selectable and there are less or equal options than defined by the threshold. In this case the procedure loops over all options UI elements and generates for each option a radio button.
- Only one option is selectable and there are more options than defined by the threshold. In this this case only one list box is generated that will be populated with data during runtime.

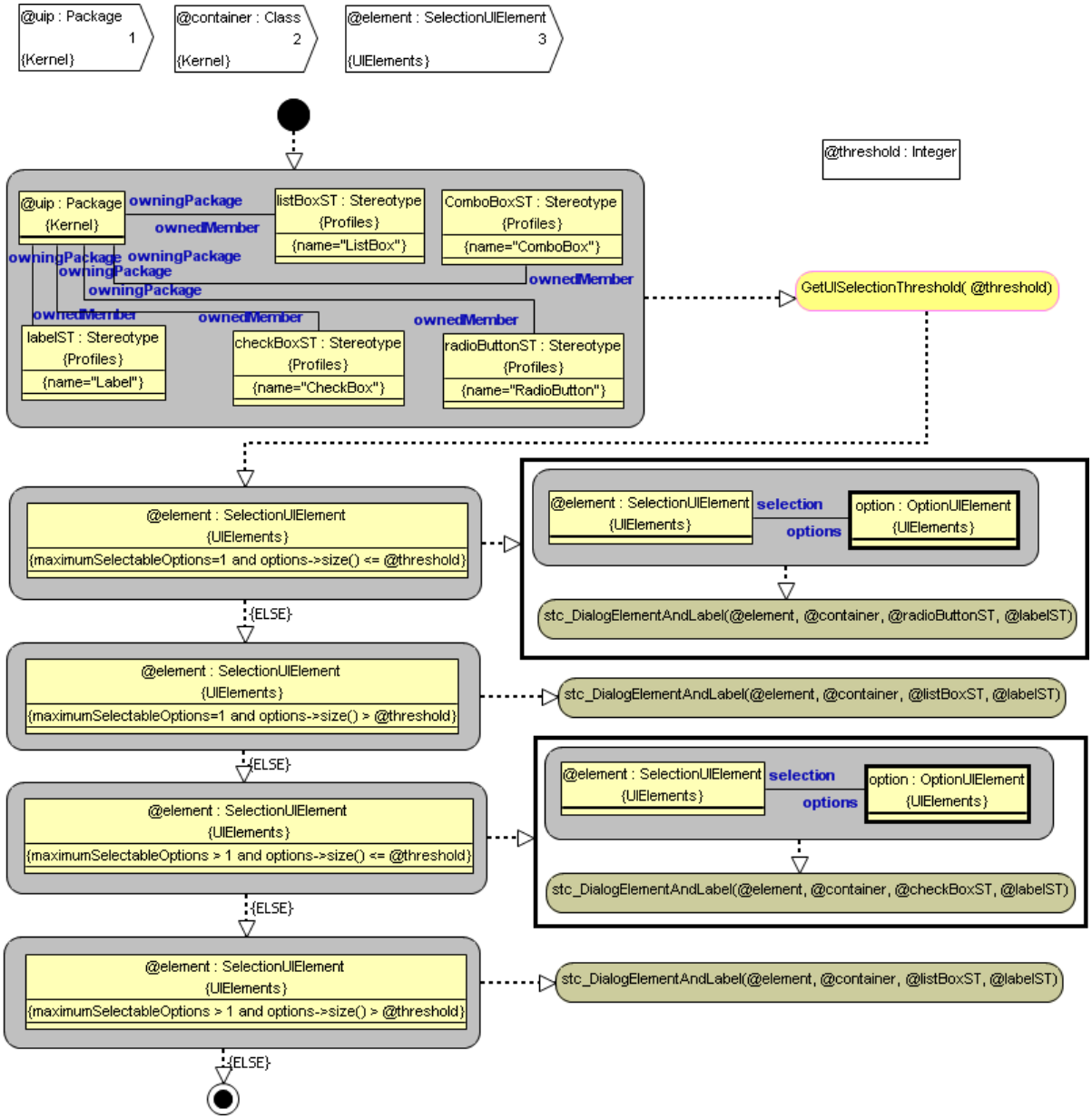

Figure 4.45: Procedure generating GUI elements for selection UI elements

- More than one option is selectable and there are less or equal options than defined by the threshold. In this case the procedure loops over all options UI elements and generates for each option a check box.
- More than one option is selectable and there are more options than defined by the threshold. In this case only one list box is generated that will be populated with data during runtime.

The procedure shown in Figure 4.46 generates either menu items or buttons for trigger UI elements. The decision if a menu item or a button is generated depends on the attached SVO sentence. If the attached SVO sentence is an initial actor predicate, a menu item is generated that invokes the dialog, otherwise a button is generated.

The procedure shown in Figure 4.47 generates the actual widget and label and adds the appropriate UML stereotype. Both widgets are added to the corresponding panel. Additionally, the procedure establishes traceability links between the UI element and both widgets (input widget and label widget). After executing this procedure for all UI elements, the complete GUI structure is generated.

To retrieve and instantiate a dialog a method is required in the UI factory. The procedure in Figure 4.48 adds a get operation for each dialog, named “get”+dialog name+“Dialog” to the UI factory. Furthermore, the procedure establishes a dependency link between the UIFactory class and the dialog class.

The procedure in Figure 4.49 adds delegate methods for each actor predicate to the UI class that implements the UI interface for decoupling the GUI widgets and the application logic. This means that all widgets forward their actions (actor predicates) to the UI class which then calls the method corresponding to the actor predicate on the application logic.

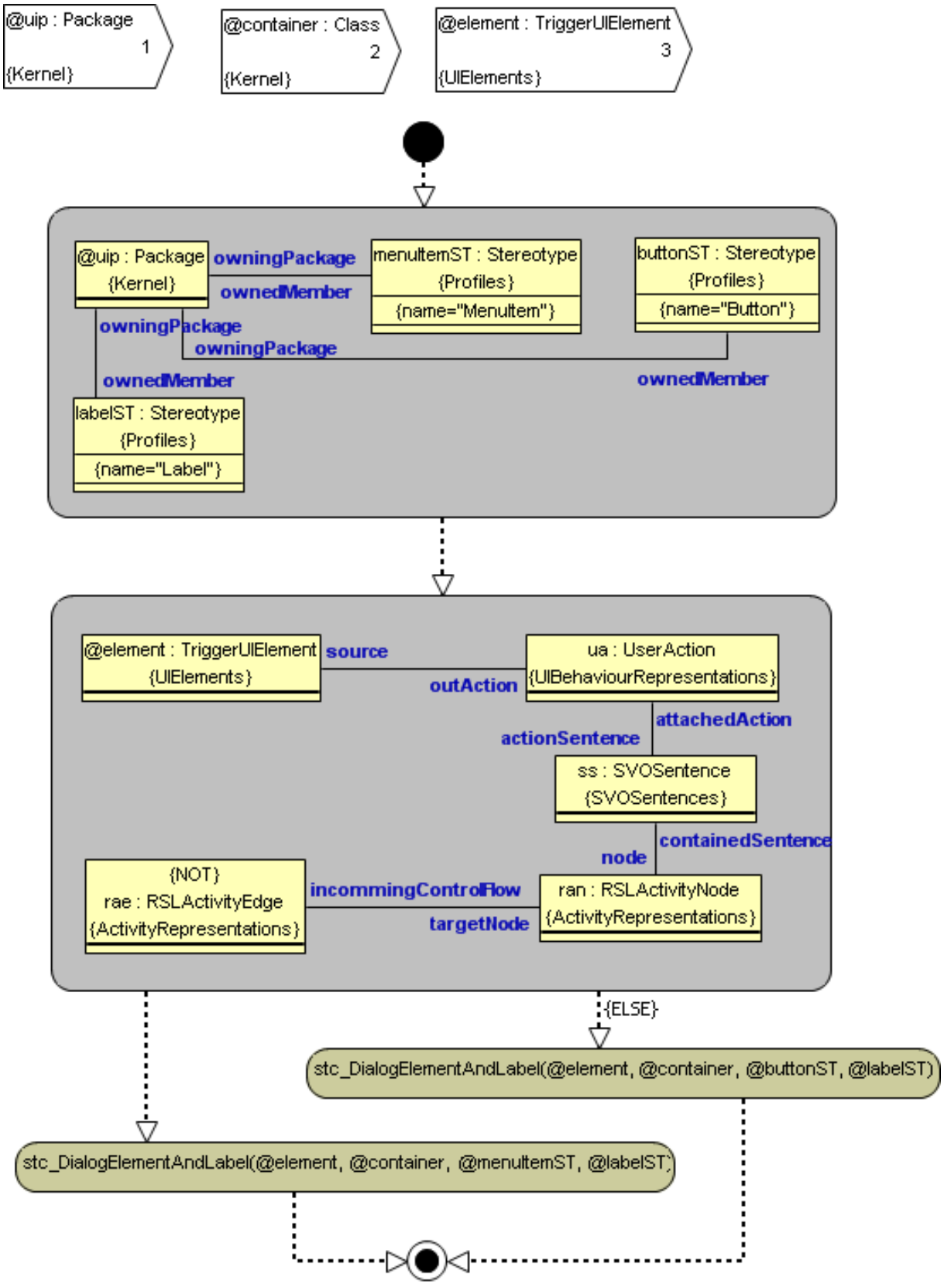

Figure 4.46: Procedure generating GUI elements for trigger UI elements

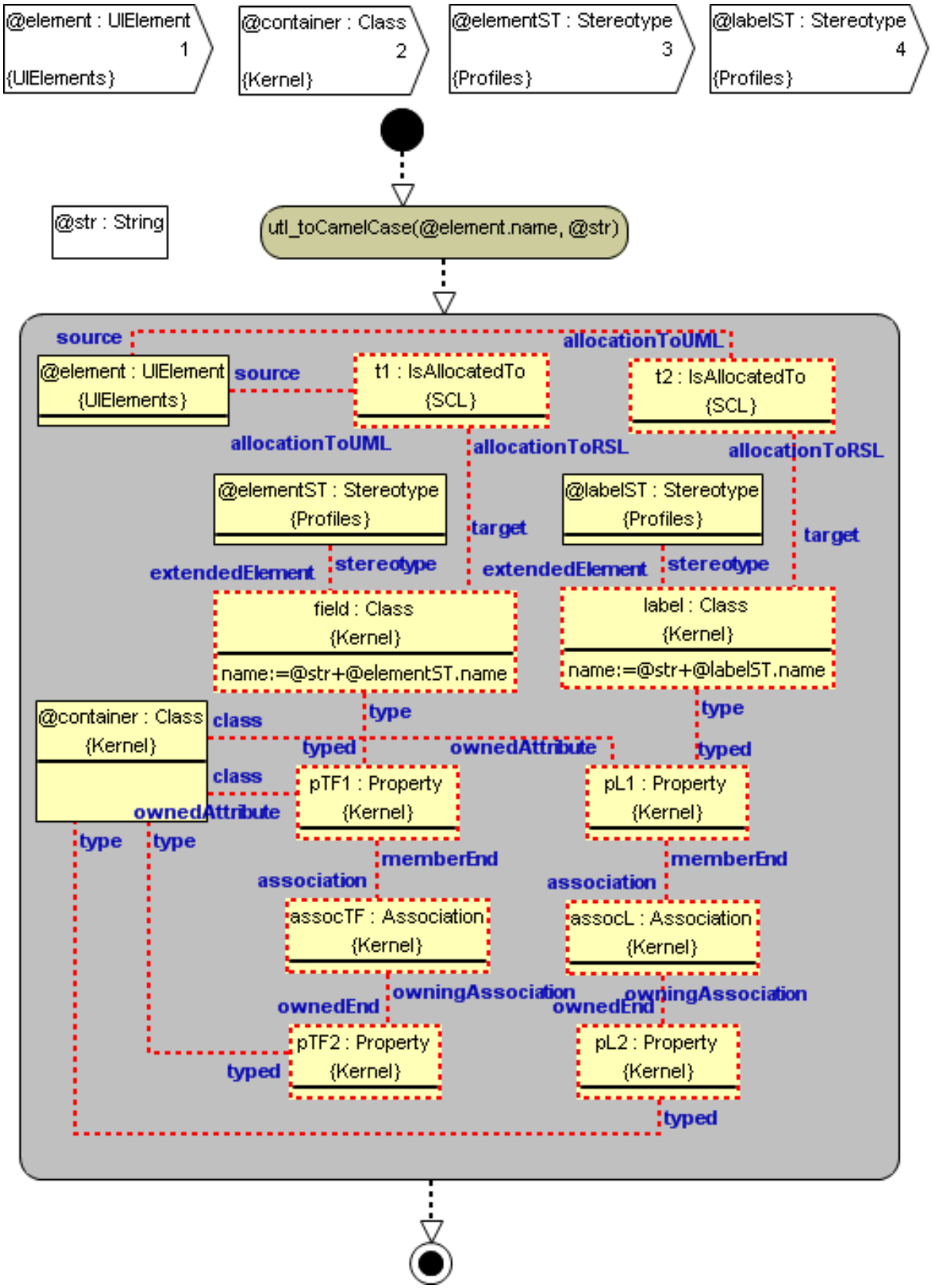

Figure 4.47: Procedure generating GUI widgets together with traceability links.

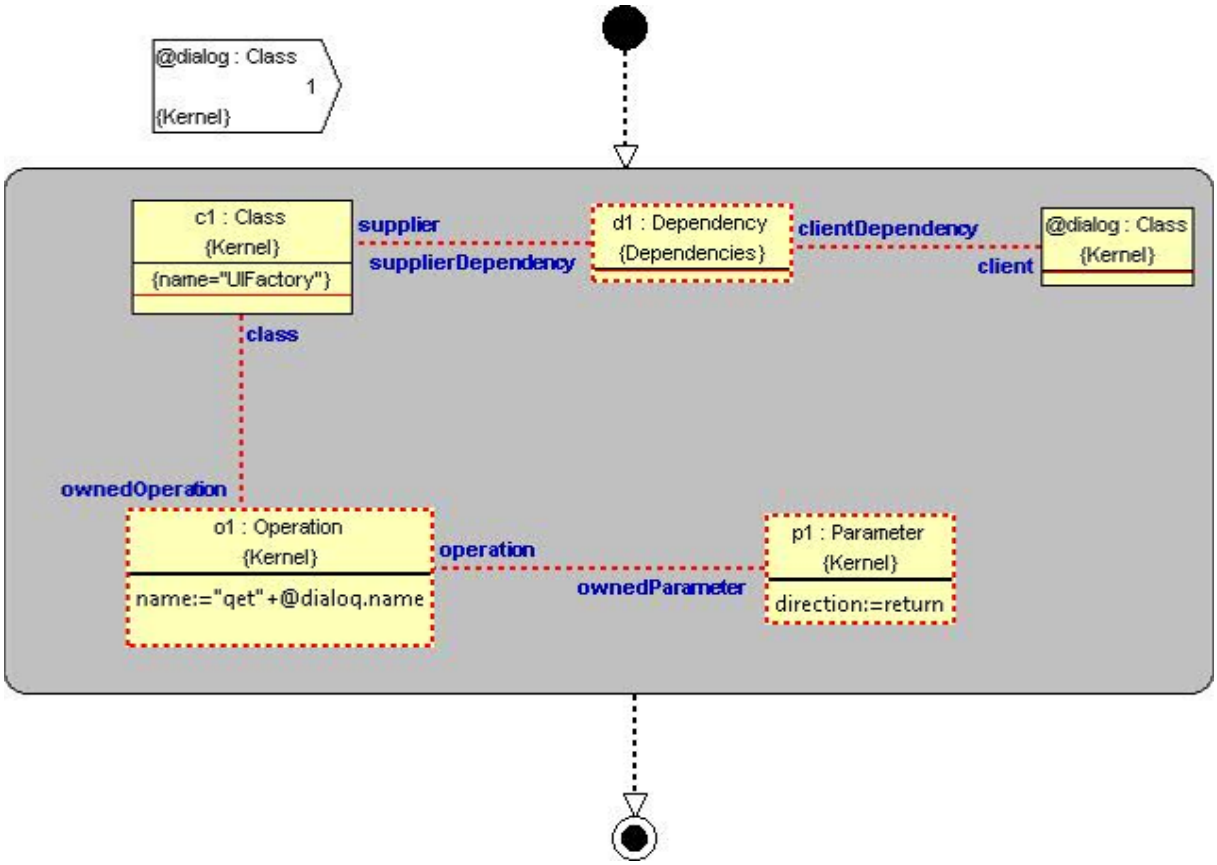

Figure 4.48: Procedure generating factory methods for retrieving dialogs.

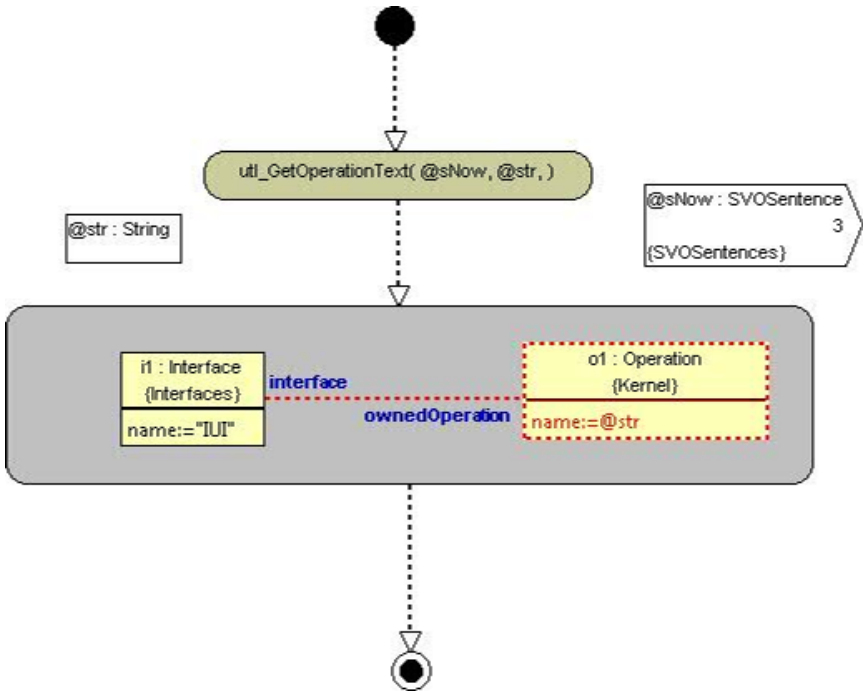

Figure 4.49: Procedure generating additional methods for the UI class.

4.4 Transformations from architecture to detailed design

The algorithms for transition from the manually refined architecture model to the detailed design, described in the section 3.4, are quite straightforward, if the chosen example methodology is used. They build the static structure of detailed design model - its packages, classes and interfaces, with the relevant operations included. Wherever possible, the dependencies between elements are built, realisations between an interface and the class realising it, and a certain amount of associations. Only, because of the fact that the 4-layer architecture is to be used, the number of elements to be generated is quite large. It is clear that without an automated transformation support, such a model would be difficult to build correctly.

The implementation of these algorithms in MOLA to a degree would remind the structure while building for architecture, but in a much more complicate way. Initially, the general package structure of the detailed design model should be built, partly, in a fixed way or partly on the basis of architecture model elements.

Then the building of classes and interfaces, their operations with full signatures and relevant dependencies can be started.

For example, to build the Application logic package, in the subpackage for the current application logic component in the architecture model, an interface and a class realisation are built for each relevant interface in the architecture, with interface operations (with full signature) copied in both.

Then for each architecture component a factory class is created, according to the given naming rules. Then for each class in this package (generated as described above) a factory operation is generated, with the name composed of the *get* prefix and the corresponding interface name, and with the return type of the corresponding class. Finally, a factory class for the whole application logic is built, with an operation for each component-related factory class.

Finally required dependencies are added - from component factory to interface realisation classes and from the top factory to component factories.

Thus the algorithm implementation in MOLA to a great degree can follow the informal description, only the correct order of instance generation must be chosen. However, taking into account the MOLA metamodel details, the number of instances to be generated by the transformation is quite large. And so will be the transformation program size.

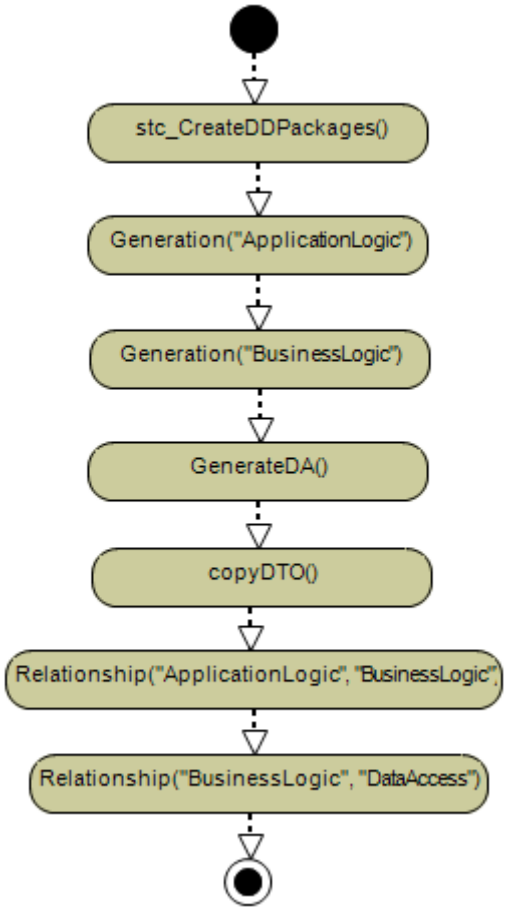

Figure 4.50: Main procedure for detailed design

Algorithms for other layers are similar.

According to the algorithm, associations (only between classes of different layers) are built according to messages in sequence diagrams in the architecture, this occurs when a class in one layer invokes an operation of an interface in another layer.

The described algorithm implementation in MOLA is quite straightforward and requires no complicated constructs, however the size of transformation procedures can be quite large. The complete implementation of these transformations will be provided later, when the issue of links to possible UML tool (Enterprise Architect) will become clearer. Namely, the necessity to see the generated classes and interfaces as readable class diagrams may impose some modifications to the algorithm implementation.

MOLA procedures implementing transformations from architecture to detailed design model are shown in figures 4.50 - 4.57.

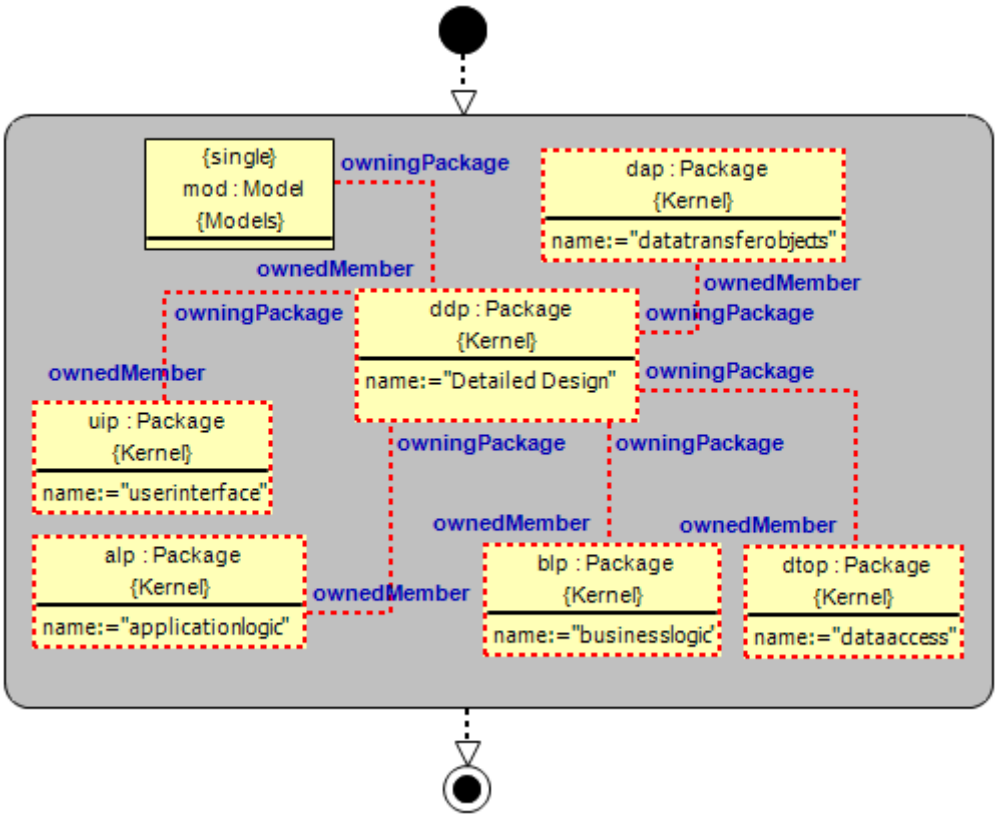

Figure 4.51: Static package creation for detailed design

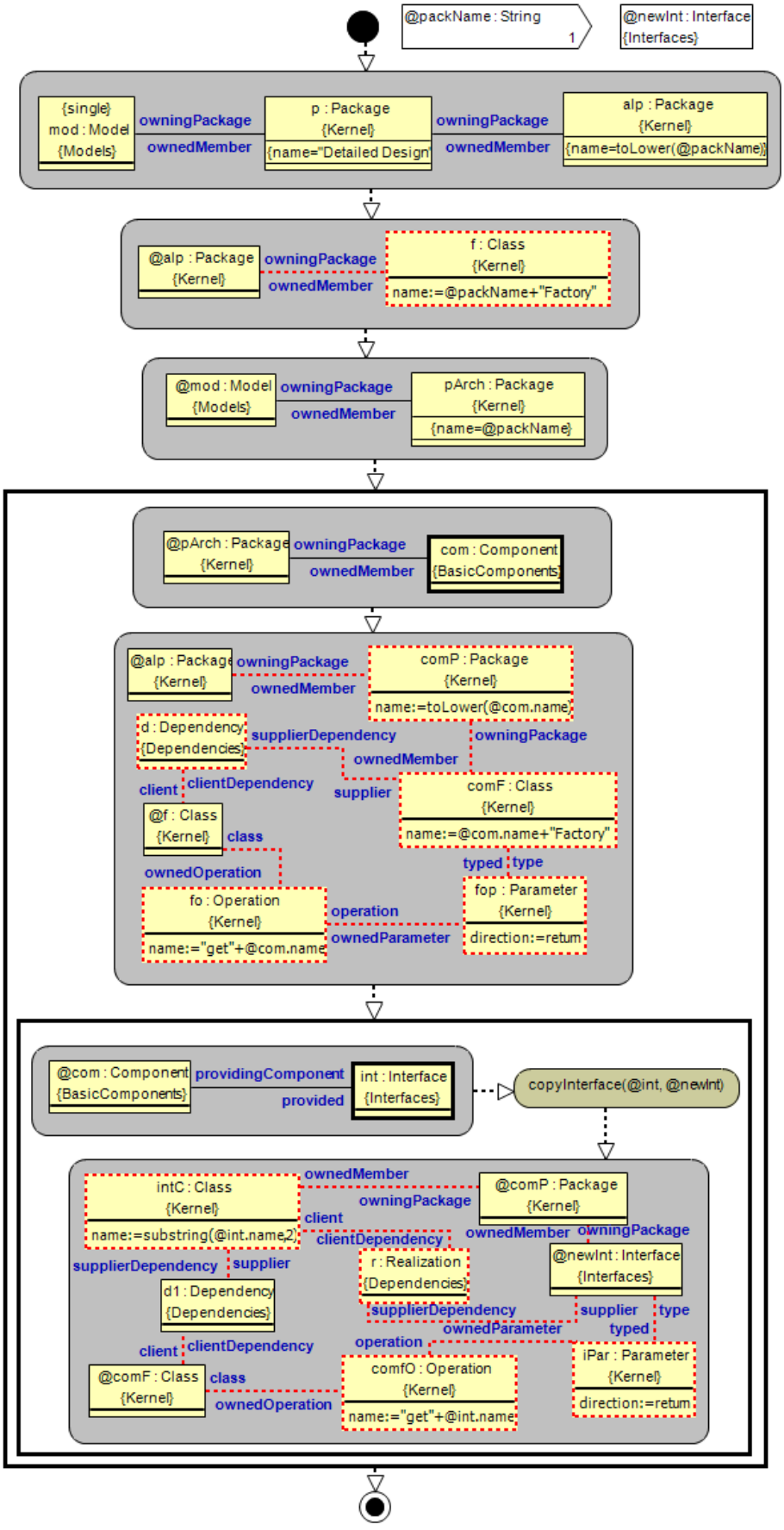

Figure 4.52: Procedure generating layers

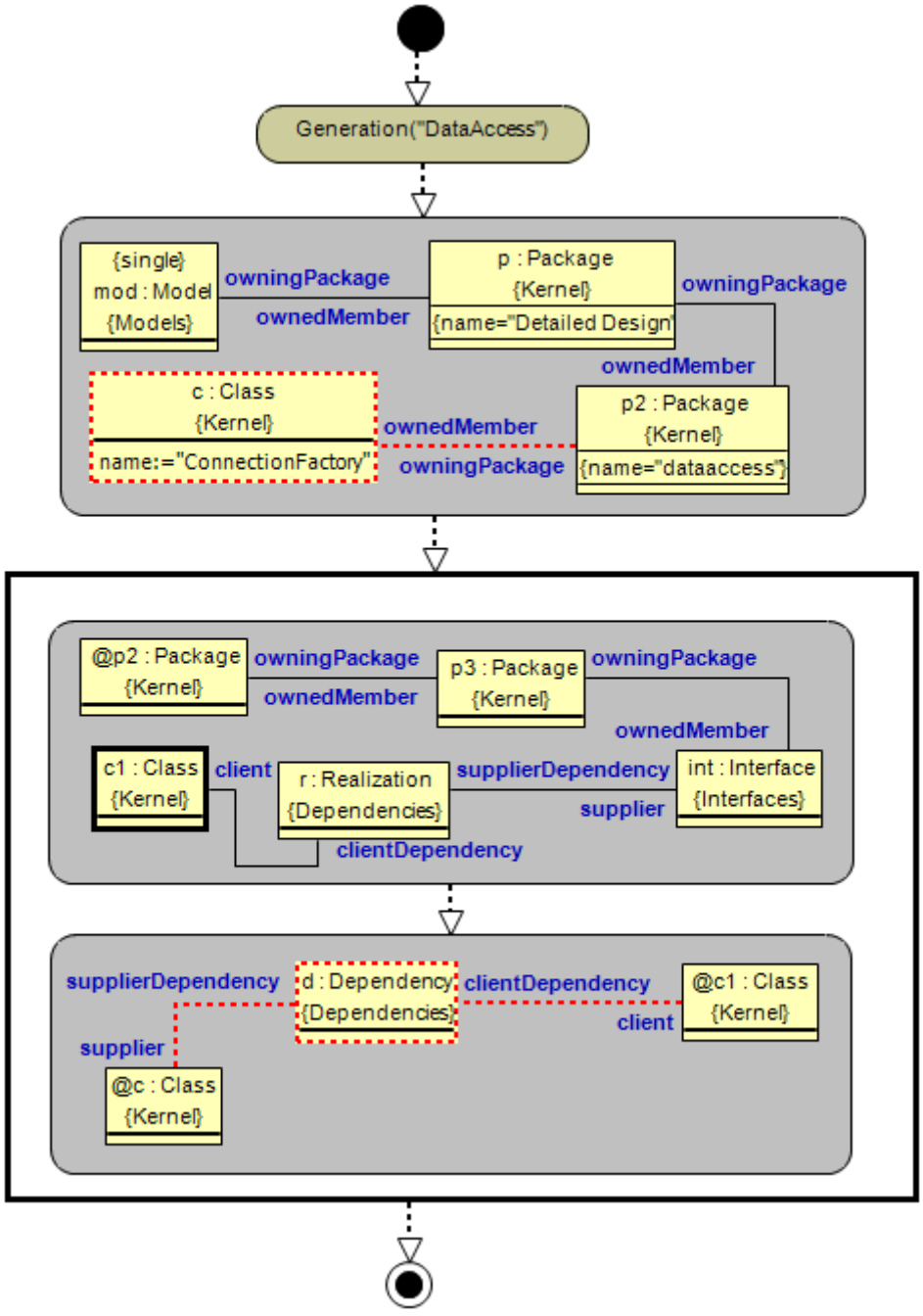

Figure 4.53: Procedure generating data access layer

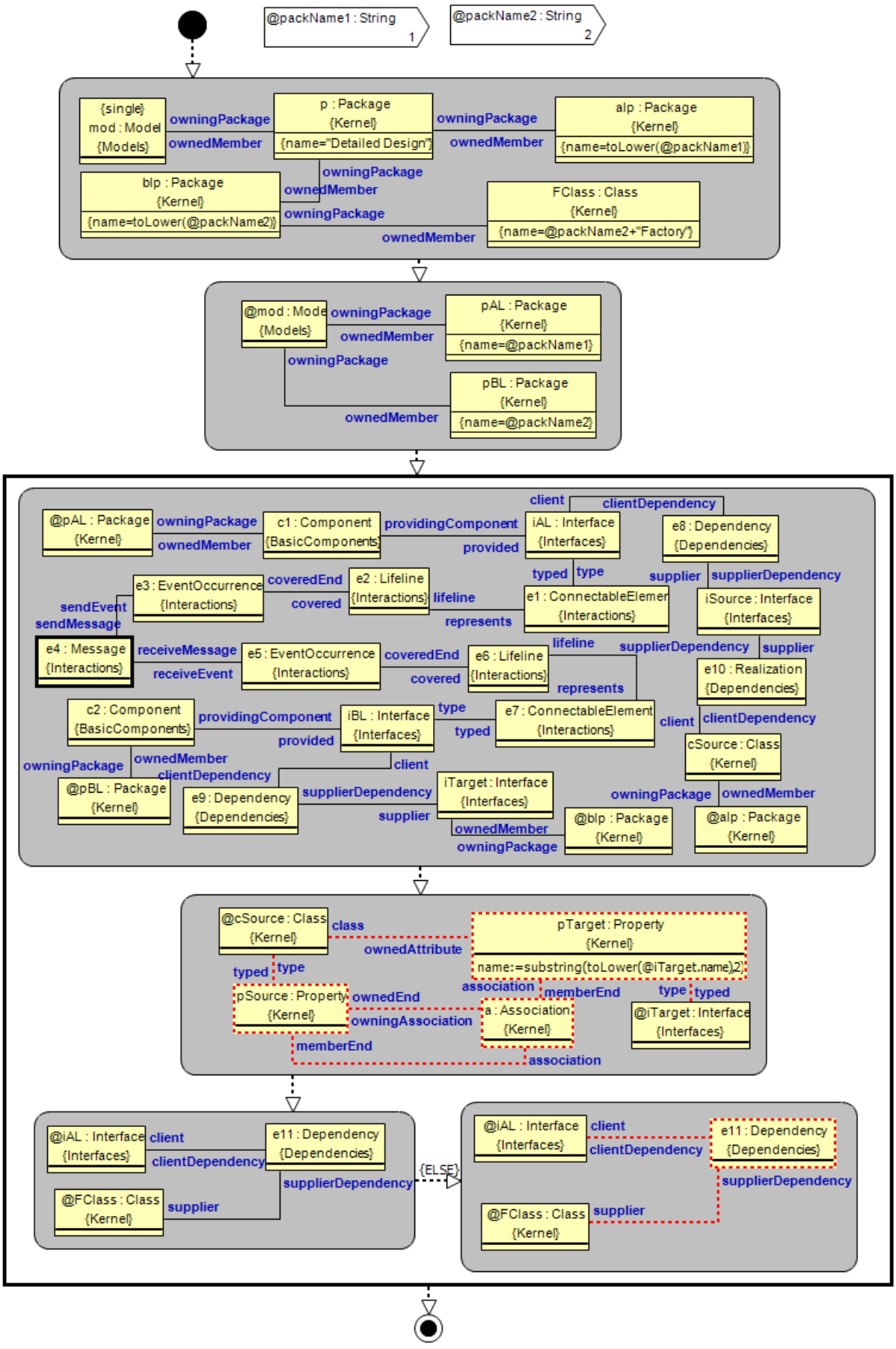

Figure 4.54: Procedure generating relationships between layers

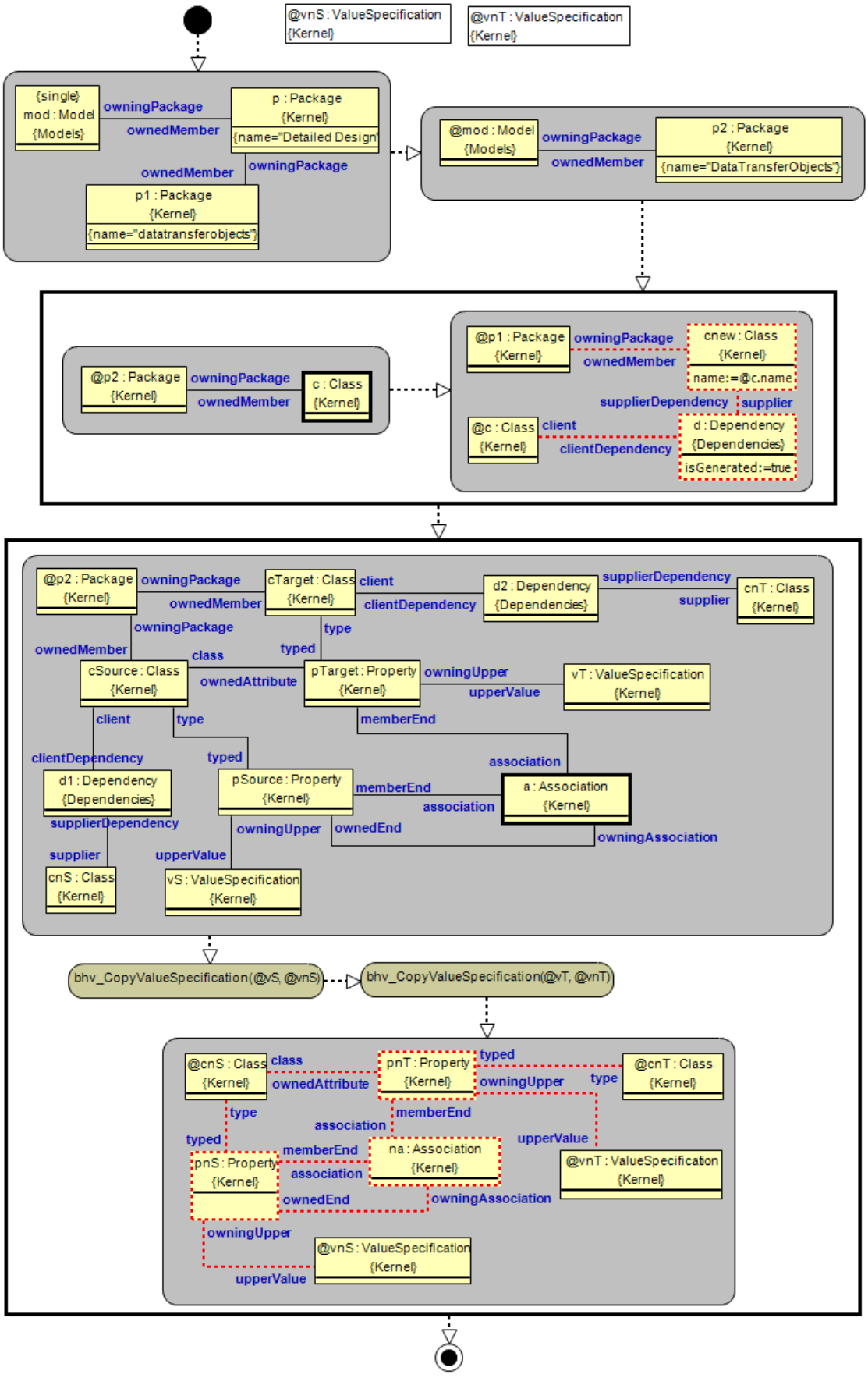

Figure 4.55: Procedure copying data transfer objects

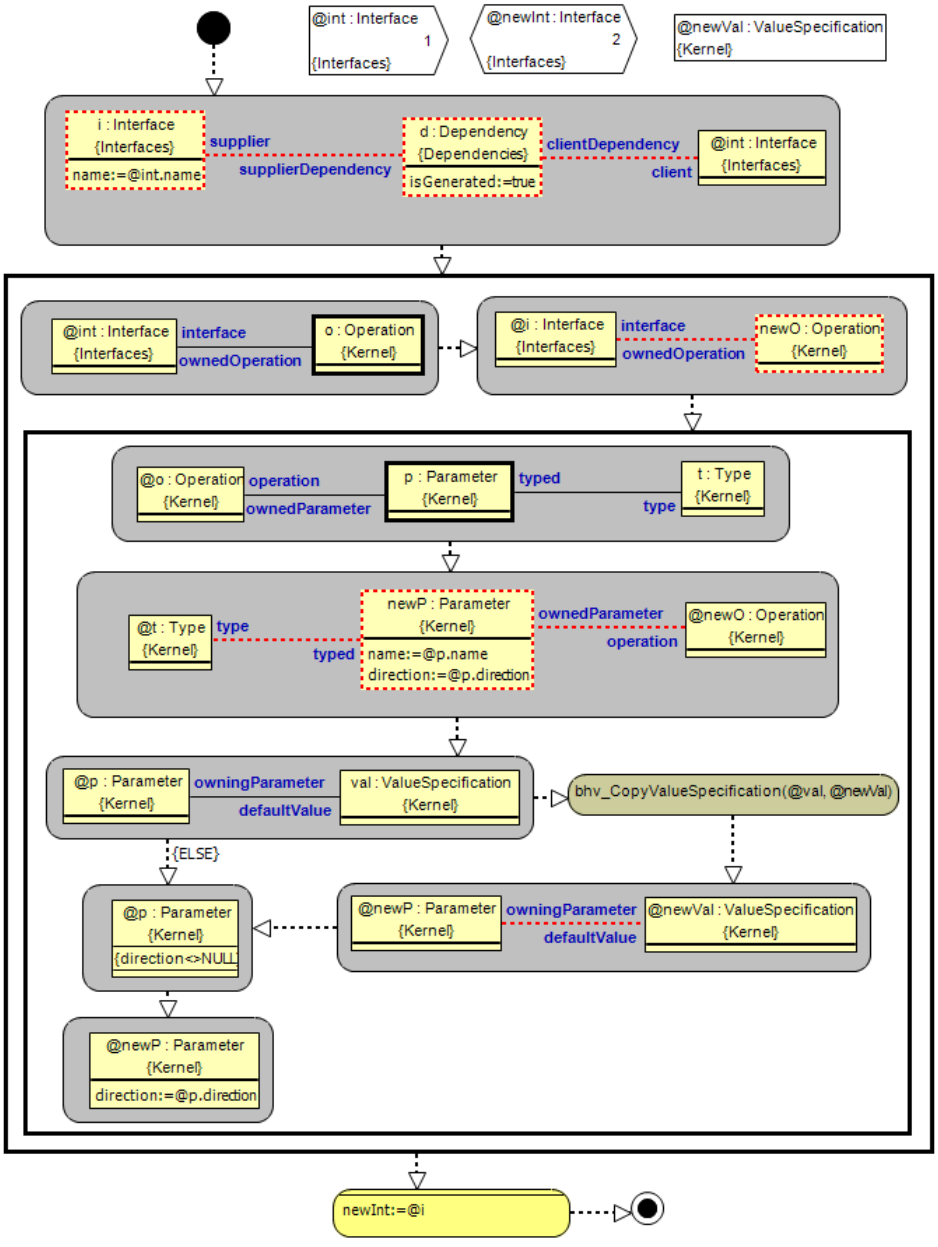

Figure 4.56: Procedure copying interfaces

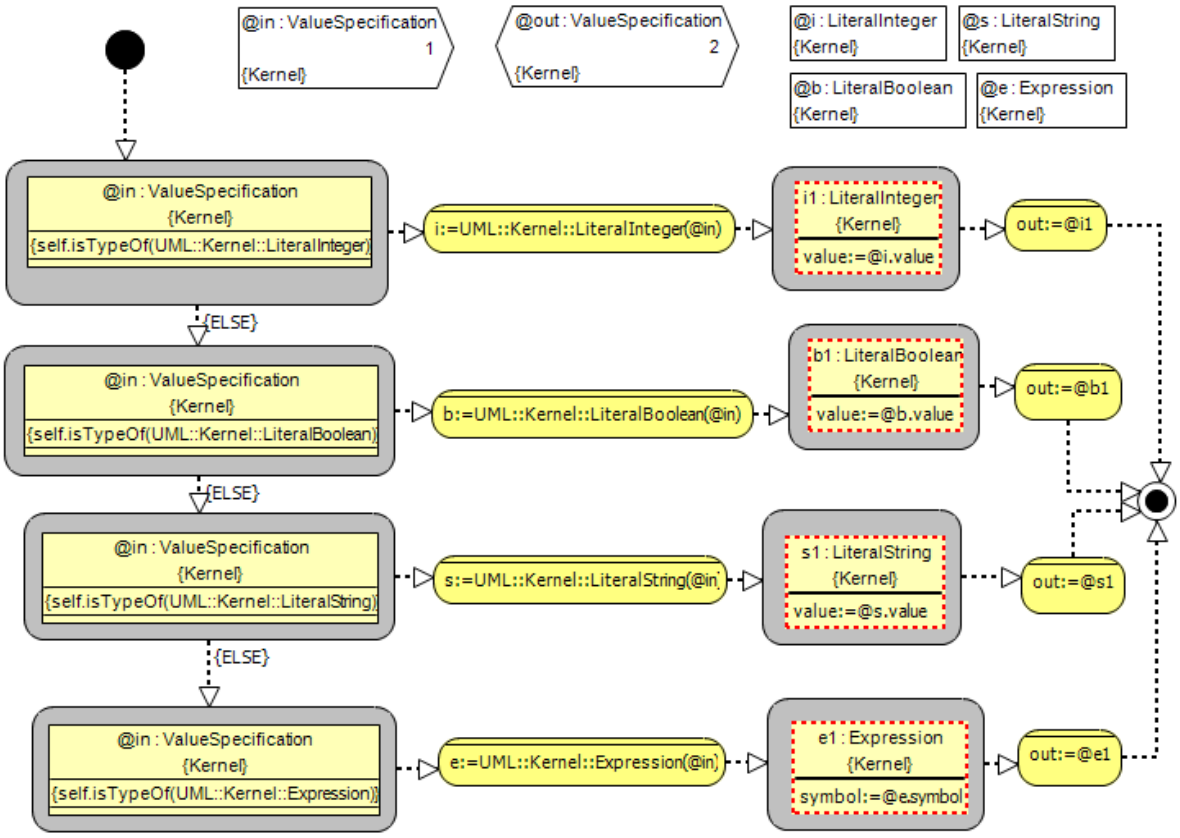

Figure 4.57: Procedure copying value specifications

Chapter 5

Providing models to be transformed

The current version of MOLA can only transform models which are in the MOLA repository. The same way, the transformation results - target models are stored in the same repository. The current version of MOLA uses a specific metamodel based repository with a well defined API (in C++).

Therefore facilities have to be developed for transferring SCL models from/to repository where they are built using the corresponding SCL editors to/from the MOLA repository. At the given stage no final decision has been made on SCL editors and repositories they store the models. However, one assumption has to be fixed already in this deliverable.

The models must be stored in their repositories *according to the tool-ready SCL metamodel*, specified already in the deliverable 3.2.1. Only in this case transformations as specified in this deliverable make sense. If models are stored by the corresponding editors according to another metamodel, then it is much easier for MOLA to use this metamodel as a source or target metamodel respectively. Especially, this refers to the RSL part of the complete SCL, where much effort has been spent on fine tuning the tool-ready RSL metamodel.

Since currently no decision has been made as to how the RSL editor in ReDSeeDS will be implemented, a temporal solution has to be provided. A prototype RSL editor (with textual representation only) is being developed by UKo partner in ReDSeeDS, using JGraLab (described in previous deliverables for software case repository implementation) as its repository. This repository directly uses the tool-ready RSL metamodel as the repository schema. Therefore this chapter mainly concentrates on the use of JGraLab as the repository, with which the source and target models are exchanged by the MOLA repository. This includes also target or source

models in UML 2.0, which are expected to be stored in JGraLab and then transferred to/from Enterprise Architect.

An alternative solution could be to use Enterprise Architect as a model repository. Then UML models could be stored in this repository in a more direct way. The Enterprise Architect repository also has a well defined internal API for retrieving and storing models. Unfortunately, Enterprise Architect as a repository does not support an arbitrary metamodel. This metamodel has to be very close to UML (the same classes and associations, only the attribute set may be extended). In addition, the metamodel version used is not UML 2.0, it is a custom version close to UML 1.3 (though externally even UML 2.1 is supported by Enterprise Architect). All this poses problems when trying to store RSL models in such a repository where this point of view is derived from the tool-ready RSL metamodel. Besides it is the software case contained design components that are required to be manipulated within Enterprise Architect by the software architect. Therefore the use of Enterprise Architect as a software case design collection repository is not discussed in more detail in this chapter.

This chapter describes model exchange solutions, which could be implemented with not large efforts and consequently, could be used for experimental application of transformations to example software models in ReDSeeDS.

The clearest and simplest problem is to import a model from the JGraLab repository to the MOLA repository, with a requirement that the metamodel to be used by MOLA is a direct subset of the metamodel used as the repository schema. Namely this is the situation with the tool-ready RSL. The situation with UML 2.0 also can be reduced to this one. The proposed solution is discussed in section 5.1. Namely this solution is the basis for some transformation testing on real instances.

The opposite way - from MOLA to JGraLab is slightly more demanding, but still manageable. It is based on the XML-RPC exchange between the tool components. The basic principles of it are described in section 5.2.

Other subsections discuss some other issues related to model exchange between repositories.

One more issue is how to exchange UML models between JGraLab (where they are placed by transformations in MOLA) and Enterprise Architect. Currently experiments show that this task is solvable, but specific solutions to a great degree depend on the chosen architecture of the ReDSeeDS tool in general. Therefore this issue is delayed to Workpackage 5.

In the conclusion one remark should be provided. It is feasible to build a new version of MOLA within ReDSeeDS, which would directly use JGraLab as it runtime repository (certainly, this MOLA implementation would be Java based instead of C++ for the current one). Then transformations could be executed directly within the JGraLab repository. Another gain would be to get rid of the single inheritance restriction in MOLA MOF, the EMOF restriction would still apply. Since this solution is more effort demanding, it could be provided only within the workpackage 5, and not for the first evaluations of transformation technology in ReDSeeDS.

5.1 Obtaining source models for transformations from JGraLab

This section describes a relatively easy implementable solution, how models from JGraLab repository can be transferred to MOLA repository, in order to be used as source models for transformations. Certainly, the metamodel used in MOLA during transformation development must be a direct subset of the metamodel used as the repository schema. However, it is feasible also to have more metamodel elements in MOLA, simply they will not get any instances.

JGraLab repository is based on TGraphs as its metamodel formalism (see deliverable 3.1, section 6.3). Currently a new simpler mapping between EMOF-based metamodel elements and TGraphs has been defined. Each metamodel class corresponds to a VertexClass in TGraphs. Each association corresponds to an EdgeClass (or CompositionClass). Class attributes correspond to VertexClass attributes, attributes of EdgeClass (having no counterpart in EMOF) are not used directly. Because of this simplified mapping model exchange has become much simpler.

The instance import algorithm is quite straightforward. It has access to the corresponding metamodel definitions both in MOLA and JGraLab repositories. The algorithm scans classes in the MOLA metamodel. For each such class a VertexClass with the same (qualified) name is sought in JGraLab. If it is found, all vertices (i.e., instances) of that class are transformed into instances of the corresponding class in the MOLA repository (attribute correspondence is straightforward). Then for each association in the MOLA metamodel the corresponding EdgeClass in JGraLab is sought on the basis of role names (EdgeClass names are generated and have no direct counterpart in MOLA), it is permitted to use more role names in MOLA than in JGraLab. For each found correspondence all instances of the EdgeClass are converted into corresponding association instances (links) in MOLA.

The only technical problem in the implementation of this algorithm is that JGraLab is in Java and MOLA repository has a C++ API. The import application (GraMol) will be developed in

Java, and it will access both repositories simultaneously. The JNI (*Java Native Interface*) will be used for accessing the MOLA repository with its C++ API. The application will directly implement the above mentioned algorithm.

MOLA transformation execution, most probably, will be invoked from the ReDSeeDS engine. Then the described import application (GraMol) will be invoked as a prologue to this execution.

This simple execution schema is oriented towards the first experimental applications of transformations in ReDSeeDS. Therefore the client-server aspect is completely ignored in this simple approach.

For storing the transformation results (target model) back to the JGraLab repository, however, a different technology, most probably, will be used. This technology is described in the next section.

5.2 Storing transformation results to JGraLab

This section explores the possibilities of storing the results of transformations performed by a compiled MOLA transformation to JGraLab. Basically, there are two concepts which can be thought of. First, the MOLA transformation could store the resulting model only to its internal repository. A separate converter tool would then have to read this model from the MOLA repository, convert it to the JGraLab format and then store it to the JGraLab repository. The other notion requires the MOLA transformation to publish the operations which must be executed on the source model in order to transform it to the target model (certainly, these operations are executed also inside the MOLA repository in the standard way, this is required to ensure the standard MOLA execution logic). These operations could then be immediately applied to the JGraLab repository.

Closely connected to the two approaches described above is the question of how to access the JGraLab repository. Besides the Java API offering the full range of graph and schema creation, manipulation and traversal, JGraLab also provides an *XML-RPC* (*XML Remote Procedure Call*) [Win99] interface for remote access. Although, at present, it only features the calling of procedures to handle graphs, this is sufficient to map the results of transformations to the repository.

The following description is based on the assumption that in both issues, the second option is chosen, i.e. the MOLA transformation directly changes the JGraLab repository via the XML-RPC interface. This alternative seems to be advantageous due to two aspects: While JGraLab is

implemented in Java, MOLA is written in C++. Consequently, a converter tool would need to employ *JNI* (*Java Native Interface*) [Lia99] in order to bridge this gap. However, the main problem with the converter approach is that no clear semantics can be defined for instance update and delete semantics in JGraLab. It should be noted, that there are no such semantics problems for instance import to MOLA. Furthermore, a current idea concerning the architecture of the ReDSeeDS engine is to run the fact repository, i.e. JGraLab, on a server while the transformation engine resides on the client side. The realisation of this architecture requires a technology to remotely access JGraLab's methods.

The next section goes more into the details of the XML-RPC interface on the server side (JGraLab). Afterwards, the client side (MOLA) is described.

5.2.1 JGraLab XML-RPC server

The server-side XML-RPC interface of JGraLab is based on the Java *Servlet* technology. Graphs and their elements, i.e. vertices and edges are referenced by using numerical handles or IDs returned by the server.

The following list shows a selection of JGraLab features which can be invoked by a client:

- creation, loading and saving of graphs
- creation and deletion of vertices and edges
- getting and setting of attribute values of vertices and edges
- traversal of graphs

As it can be seen, these features cover all the operations needed to perform changes as required by a model transformation: creation, deletion and modification of graph elements. The latter corresponds to the modification of attribute values.

5.2.2 MOLA transformation XML-RPC client

As it was already noted, to perform this simultaneous modification of JGraLab repository, the MOLA transformation for each repository related operation (instance creation, instance deletion, attribute modification, link creation/deletion) must perform the same operation on the

JGraLab repository. Since initially the MOLA repository was a copy of part of JGraLab repository, this synchronous execution of operations would guarantee, that at the transformation execution end also the JGraLab repository would contain the target model.

Certainly, to achieve this, the MOLA compiler has to be extended to perform these duplicate operations on the JGraLab repository, whenever there is such an operation in the native repository. The size of extended code seems not to be very large.

These special operations will invoke the relevant JGraLab API operations via XML-RPC. Only the graph (i.e., instance) modification operations are required in this approach. All these operations are supported by the XML-RPC client.

The main gain from this approach would be the completely safe instance management semantics in JGraLab.

5.3 Storing of traceability information

The transformation of models requires traceability links to be created between corresponding elements of the source model and the target model. For example, a Requirement specified by an RSL model is connected to the architectural UML construct supposed to realise the Requirement. This connection is established by a traceability link of type IsAllocatedTo. A design model having its source in an architectural model is linked to the latter via an UML::Dependency. Generated program code can be traced back to its source model by Implements links. See Deliverable 3.2.1 for more details on different traceability link types.

The JGraLab repository holds a graph representation of artifacts contained in a Software Case. Thus, it is also well suited to store information on traceability links as additional edges interconnecting the representation of related artifacts. Since the transformation engine is supposed to create instances of traceability links along with the target model, these links could be stored to the repository according to the concepts described in section 5.2.

Chapter 6

Conclusion

This document describes transformations to be used for software case development in ReDSeeDS. Possible generation of the initial version of the next model from the previous one is covered for the basic transition steps in software case development.

This document offers one specific 4-layer architecture and methodology elements related to it, where the part of the next model to be generated automatically is sufficiently high. If a software case is being developed according to this architecture then the document provides ready-to-use transformations from requirements in RSL to architecture model in a subset of UML and from architecture to detailed design, as well as transformations for building user interface elements. Both the informal transformation algorithms and their implementation in MOLA language are provided in the document. Therefore the transformations can be used as is or easily be modified if the software case requires some modifications to the architecture.

The transition step from the detailed design model to code is not covered in this document since this requires some basic solutions in ReDSeeDS tool set to be made beforehand.

The document discusses also basic principles, which should be used for obtaining models to be transformed from the software case repository and placing the transformation results back to this repository. Currently only the version where models are taken from a JGraLab based repository and placed back to such a repository is discussed. This version could be implemented relatively easily. However, the final solution in this area also depends on some general architecture principles for the ReDSeeDS tool set.

Bibliography

[KSC+07] Audris Kalnins, Agris Sostaks, Edgars Celms, Elina Kalnina, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Volker Riediger, Hannes Schwarz, Daniel Bildhauer, Sevan Kavaldjian, Roman Popp, and Jurgen Falb. Reuse-oriented modelling and transformation language definition. Technical report, ReDSeeDS, 2007.

[KWW03] A G Kleppe, J B Warmer, and Bast W. *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, 2003.

[Lia99] Sheng Liang. *The Java Native Interface. Programmer's Guide and Specification*. Addison-Wesley, 1999.

[MM03] Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide Version 1.0.1, omg/03-06-01*. Object Management Group, 2003.

[Obj05] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04*, 2005.

[Obj06] Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, 2006.

[SCG+05] P Swithinbank, M Chessell, T Gardner, C Griffin, J Man, H Wylie, and L Yusuf. *Patterns: Model-Driven Development Using IBM Rational Software Architect (IBM Redbook SG24-7105-00)*. International Business Machines Corporation (IBM)., 2005.

[SV06] T Stahl and M Voelter. *Model-Driven Software Development (Technology, Engeneering, Management)*. John Wiley & Sons, Ltd, 2006.

[Win99] Dave Winer. XML-RPC Specification, June 1999. http://www.xmlrpc.com/spec.