VRIJE UNIVERSITEIT BRUSSEL – FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND APPLIED COMPUTER SCIENCE
SYSTEM AND SOFTWARE ENGINEERING LAB

UML - SDL ROUND-TRIP ENGINEERING THROUGH INCREMENTAL TRANSLATION OF CHANGES

Kurt Verschaeve

February 2001

Advisor:
Prof. Dr. Viviane Jonckers

A dissertation in partial fulfillment of the requirements of the degree of Doctor in Sciences.

VRIJE UNIVERSITEIT BRUSSEL – FACULTEIT WETENSCHAPPEN
VAKGROEP INFORMATICA & TOEGEPASTE INFORMATICA
LABORATORIUM VOOR SYSTEEM EN SOFTWARE ENGINEERING

UML - SDL ROUND-TRIP ONTWIKKELING DOOR HET INCREMENTEEL VERTALEN VAN WIJZIGINGEN

Kurt Verschaeve

Februari 2001

Promotor:
Prof. Dr. Viviane Jonckers

Proefschrift voorgelegd tot het behalen van de wetenschappelijke graad van doctor in de wetenschappen.

Abstract

In the real-time and telecom business, SDL is pre-eminently the language for design and implementation. For many good reasons, UML is put forward as the front-end language for analysis and system design of SDL based systems. UML and SDL are compatible as they share a static definition of structure with a behavioral definition of active entities with state diagrams. Still, each of them provides enough advantages over the other to make it worthwhile to combine them in a single methodology.

To make the combination of UML and SDL successful, it is essential to have round-trip support that synchronizes the information common to the UML model and SDL specification. The UML model and SDL specification provide a distinct view on a different abstraction level on the same system. Changes on either level must be applied to the other level if the change concerns common information. However, the mapping between UML and SDL concepts of structural and behavioral information is not one-on-one. We propose a complex mapping that overcomes the existing incompatibilities and takes into account the role of UML and SDL in the methodology. This complex mapping makes building a translator hard and makes applying existing round-trip approaches very difficult.

In this thesis, we present a set of translation rules that define how changes in a UML model are translated into changes in the linked SDL specification and the other way-around. For example, a new UML class with stereotype «block» is translated by creating a new SDL block; or renaming an SDL variable is translated by renaming the corresponding UML attribute. Each rule detects a specific change and translates it by making the appropriate change on the other side. Corresponding entities in UML and SDL are linked with each other such that changes may be applied at the right place. A continuous thread throughout all translation rules is to preserve as many manual changes as possible. For example, if the stereotype of a class changes from «process» to «block», the original SDL process is preserved and is placed in the new SDL block. Similarly, renaming an UML entity will result in renaming the linked SDL entity, without altering its contents.

Based on the incremental translation rules, we provide the full process for forward iteration and reverse iteration. Together they provide full UML-SDL round-trip engineering. The first step in the forward iteration process is building the internal information model of the new and the old UML model and of the SDL specification. The information models are streamlined toward the translation and are extended with UML-SDL link information. Next, both UML models are preprocessed for comparison and translation. Some important features of the preprocessor are calculating the inherited association, flattening nested state diagrams and assigning default stereotypes to classes and operations. Then the two UML models are compared to find new, deleted and matching entities. Finally, each change is translated by executing the corresponding set of translation rules. The attributes of matching entities are further compared as part of the translation rules.

Acknowledgements

I am most grateful to my supervisor Viviane Jonckers for all the years of support, feedback and invaluable discussions. From the very beginning, she gave me the possibility to perform research in the best possible environment. I am grateful for the opportunity she gave to me to perform research, to travel to conferences and foreign project partners and to finish this PhD.

Special thanks to Bart Wydaeghe for being a fine colleague and a best friend at the same time. Our discussions were ever enlightening and inspiring and definitely contributed to the results in this work. Our non-academic conversations made work more enjoyable and over the years, I have taken over his positive way of thinking.

I wish to thank all my colleagues and former colleagues at the System and Software Engineering Lab for provinding a great and motivating work environment. In particular, I thank Luc Goossens for all the interesting discussions and for answering many questions on various topics. Many thanks to Ludo Cuypers for leading me and Bart Wydaeghe into the INSYDE project and providing us with a fruitful start such that we could succesfully continue the project. I also thank Wim Vanderperren for his talented work during his internship.

Of particular help from Telelogic were Anders Ek, Niklas Landin and David Prather. I want to thank Anders for guiding me into Telelogic and for our initial work on the UML to SDL mapping. Thanks to Niklas for the nice cooperation and fine-tuning the translation. Thanks to David for his continuous technical support.

It is the encourgements from many people that convinced me to start, continue and finish this PhD. I am grateful to my father, who triggered me for doing research and helped me through the sparse moments of doubts. Thank you Serge Demeyer for persistently pushing me toward a concrete PhD and for the early review and helpful hints on improving the text. Last but not least, thanks to my spouse Pascale for all the support and encouragement and for granting me the most wonderful gift in life, our son Jeroen.

“Every large system that works started as a small system that worked.”

Anonymous

Table of Contents

List of Figures

I. THE THESIS

"You must have long term goals to keep you from being frustrated by short term failures."

- Charles C. Noble-

I. 1 Contribution

I 1.1 Problem Statement

The Specification and Description Language (SDL) [ITU94] is an important real-time software engineering language with a wide range of applications such as telecommunications, aerospace and automotive. SDL has a rich grammar to describe behavior and the semantics are formally defined. As a result, SDL tools can simulate systems specified in SDL and allows detections of errors in the specification at a very early stage. However, SDL is less suited for the first stages in the development life-cycle because of its lack of generic concepts and modeling freedom. Many real-time developers are turning to object-oriented analysis for the first stages in their development life-cycle.

Object oriented analysis techniques provide a good medium for visualizing, constructing, describing, and documenting the artifacts of software systems. The different diagrams available in object oriented modeling languages (OOMLs) each cover another aspect of the system and the underlying model can be presented in different ways to clarify specific relations. This is also a way to handle complexity, as complex diagrams can be divided into smaller ones. Finally, because OOMLs in general give little constraints during modeling, is gives the system architect more flexibility to explore solutions. However, the same characteristic that make these modeling languages good for analysis, makes them poor for excact specification of a system's dynamic structure and behaviour.

The solution is to use a OOML for analysis and the first stages of design and to use formal design techniques for design and implementation. A number of projects, methodologies and tools realized by important players in this field confirm the trend towards this combination. The following projects all combine OMT and SDL in their methodology: INSYDE [HWW96], SISU I & II [Bræ96] & [MS97], SCREEN [SCR98] and TOSCA [TOS98]. The Integrated Method (TIMe) uses the unified modeling language (UML) for object nodels in early phases, SDL for design of structure and behaviour and message sequence charts (MSC) for describing interaction. The tool Telelogic Tau offers extensive support for both UML and SDL.

A recurring problem when combining different models on different abstraction levels is the synchronization of overlapping information. After the initial translation step, the development continues by adding more details to the generated design. When turning back to the higher abstraction level, e.g. to make structural changes, simply rerunning the translation results in overwriting the previous changes. Relevant changes on one level of abstraction should automatically be reflected on the other abstraction level while preserving as much of the detailed design as possible. Without the necessary tools support for round-trip engineering, the higher level design models will not get updated and loose their effectivness or will not be created in the first place.

In short, we need a round-trip engineering process that integrates UML for system design and SDL for detailed design and synchronizes the common aspect of the UML model and SDL specification in such a way that previous detailed handwork is not overwritten.

I 1.2 Contribution

The major contributions of this thesis are to bring the UML and SDL languages closer to each other and to enable round-trip engineering based on the complex mapping between those languages.

In this thesis, we present a validated mapping between UML and SDL concepts. In short, the UML class diagrams map on the SDL static structure and the UML state diagrams map on SDL state charts. The mapping is complex because entities do not map one-on-one. For example, a UML association is mapped on a combination of channels, signal routes and/or gates and a class can be mapped on a block and/or process. Moreover, some incompatibilities need to be resolved, e.g. nested state diagrams must be flattened before they can be translated.

Based on the complex mapping, we provide support for round-trip engineering between a UML model and a SDL specification. The typical one-shot translation is replaced by a set of incremental translation rules that translate changes rather than complete models. The model and the specification are stored internally in an information model for UML and SDL that is specialized toward the translation. The UML model is preprocessed to fill-in missing information, to check consistency and to overcome two incompatibilities with SDL. More specifically, inherited associations to subclasses and nested state diagrams are flattened.

The incremental translation algorithm is based on finding the changes made since the previous iteration and translating only those changes. To find these changes, the model is compared with its previous version based on the entities' unique identifier. The result is a set of new entities, deleted entities and matched entities. Matched entities are entities that are present in both the old and the new model and their attributes are further compared as part of the translation. This comparison is executed in a hierarchical fashion.

We developed a large set of translation rules that translate any possible change in UML (e.g. new class, rename association, delete attribute) to local changes in SDL and the other way around in complience with the mapping. Each translation rule consists of a combination of preconditions, context, translation actions and variable definitions. Moreover, the translation is always done in such a way that as much of the detailed design work in SDL is retained. For example, if a class with stereotype «block» becomes "typed", the corresponding block is converted to a block type, takeing over the complete contents of the block. This approach of updating the model instead of regenerating it also makes sure that the graphical layout information of SDL entities is retained.

In order to determin the correct location to apply the translated changes, we extended the UML and SDL information model with explicit links between UML entity and SDL entities. When translating a "new" UML entity, it is linked with the generated SDL entities and some additional links needed to translate other change. A «block» class, for example, is linked with the generated block (*sdldefinition*), the management process (*sdlprocess*), the structure containing the signals (*declarationStruct)* and the signal list (*sdlsignallist*). The reverse version of these links are added to the SDL information model, e.g. *process.sdlprocess*$^{-1}$ returns the class that is linked to the process though the *sdlprocess* link.

In short, we introduce the technique of incremental translation of changes to synchronize the UML model and the SDL specification, while they present different abstraction levels of the system. The rest of this section motivates why this contribution is relevant, novel and important to the telecom community.

I 1.3 Relevance

There is an increasing demand for an integration of object-oriented analysis with formal design techniques for the development of real-time and embedded systems. Many real-time developers are turning to object-oriented analysis for the first stages in their development life-cycle. It is a natural demand to integrate this into the rest of the development life-cycle. This is reflected in a number of projects, methodologies and tools realized by important players in this field.

The INSYDE methodology [HWW96] integrates the object-oriented method OMT with two domain-specific design techniques, namely SDL [ITU94] and VHDL. This research was performed by three academic and three industrial partners. Baseline for the process model is OMT, which offers a common platform for the analysis of software and hardware. The software parts are translated to SDL and the hardware parts are translated to VHDL. Detailed design continues in the specific target language. This combination of complementary notations offers a number of advantages.

Octopus is a systematic and effective method for developing object-oriented software particularly for embedded real-time systems. It has been developed at Nokia Research Center since 1993. The Octopus method is based on the popular OMT and Fusion methods, but also embodies common practice found in real-time system development.

The methodologies developed by the SCREEN [SCR98] and TOSCA [TOS98] projects are based on components being developed by combining UML (or OMT) and SDL. These components are then incorporated into a framework. COSEC and TOSCA have in common that they provide a rapid service provisioning based on the specialization of a framework with a nearly ready service set of software components. The framework can be used to build a large number of standard and customized services. More details about the alignment of the TOSCA and SCREEN approaches can be found in [LKH99].

Another notable example is TIMe, The Integrated Method, which is a systems development methodology from SINTEF Telecom and Informatics. TIMe is based on over 2 decades of experience from research and industrial projects. It uses the unified modeling language (UML) for object nodels in early phases, SDL for design of structure and behaviour and message sequence charts (MSC) for describing interaction.

Within the SDL community, UML has gotten a lot of attention the last few years. The latest version, SDL 2000, has a build-in graphical notation for the UML class diagram. Both major SDL tool builders, Telelogic and Verilog, are building tools to integrate UML and SDL. Telelogic Tau 3.6, now covers all the phases of the development process and covers them with languages optimized for each phase: UML, SDL and TTCN. We actively contributed to implementing the translation of UML to SDL.

Even in an integrated tool/language, there will still be the need to maintain two levels of abstractions. The system design view, is used for documentation and overview and on this level, important design and architectural decisions are taken. On the implementation level, all the details are accessible. In this scenario, support for round-trip engineering is still necessary to synchronize the different abstraction levels.

I 1.4 Novelty

At the time we first developed translation rules from OMT to SDL'88 [VWCJ95], this was a brand new idea. The gap between the two languages was very wide and the OMT semantics had to be bent a lot to make the translation to SDL. Especially the integration of OMT's static model and the dynamic model in the SDL code generator was novel. The transition to UML and SDL'92 imposed new possibilities and challenges. A continuous process of improvements [Ver97], partially in cooperation with Telelogic [VE99], brought the quality of the translation good enough for commercial exploitation.

The integration of the class diagram and the state diagram of UML for round-trip engineering is new. UML tools that support round-trip engineering only translate the class diagram [Tog00], [Rat00]. Methods that translates both the class diagram and the state diagram [Har97], [Mel99] are a one-shot translations and require to write a lot of code in the state diagram.

Round-trip engineering with complex underlying translation rules is new. The translation of UML to SDL is complex; there is no simple one-to-one mapping between an entity in UML and an entity in SDL. For example, an association maps on a set of channels and signal routes, spread over several structures. If the association is modified, this may have implications on all SDL entities generated from the association. As long as there is no one-to-one mapping between UML and SDL, an incremental translation of changes is a good option to synchronize these abstraction levels. Tools like TogetherJ [Tog00] that do UML-Java round-trip engineering, are based on an exact one-to-one mapping.

I 1.5 Importance

For developers that already use UML and SDL, it is of course very important that their process is supported by the right tools and methods. Moreover, tool support for UML–SDL round-trip engineering can really boost object-oriented design, with all its advantages, into the whole SDL community. Automatic synchronization encourages people to maintain and exploit the system design model of the system. This is already important in a waterfall-like process, but is crucial in an iterative software development process.

Within the SDL community, UML has gotten a lot of attention the last few years. The latest version, SDL 2000, has a build-in graphical notation for the UML class diagram. Both major SDL tool builders, Telelogic and Verilog, are building tools to integrate UML and SDL. Telelogic Tau 3.6, now covers all the phases of the development process and covers them with languages optimized for each phase: UML, SDL and TTCN. We actively contributed to implementing the translation of UML to SDL.

Even in an integrated UML/SDL tool/language, there will still be the need to maintain two levels of abstractions. The system design view, is used for documentation and overview and on this level, important design and architectural decisions are taken. On the implementation level, all the details are accessible. In this scenario, support for round-trip engineering is still necessary to synchronize the different abstraction levels.

This research is also reusable outside the UML-SDL scenario. Providing round-trip support by incrementally applying translation rules can be applied to synchronize any two models on different abstraction levels that contain parallel information. The only prerequisite is a set of translation rules that is able to translate the abstract model into the concrete model in an entity per entity fashion. These translation rules are slightly modified to translate new items added after an

iteration. To translate other changes, the set of translation rules must then be extended with rules that translate any individual change in the model.

I 1.6 Larger Research Context

The research done for this dissertation has been part of several broader research efforts. During the INSYDE project [INS94], OMT was used as an analysis and design front-end to SDL and VHDL. The common front-end enabled developers to co-design hardware and software in one method. In the ITA-2 [ITA98] AIA project [AIA98], we do research on a methodology for component oriented service creation [VWW00]. One specific feature of the methodology is that it targets different kind of users, ranging from developers who demand high-flexibility to end-users who like ease-of-use. In this research, the UML-SDL round-trip engineering is used to build an SDL component framework with an easy-to-use UML front-end.

The results of our research should be applied in a larger context. In this dissertation, we provide only a part of a software development process. In order to use the UML-SDL round-trip engineering in a real project, it should be fit into a global iterative process. Because such a process is (and should be) different for each company or even for each project [Hig00], we do not put forward a preferred process in this dissertation.

The core of our research, automatic synchronization of models expressed in different paradigms, can be applied to other languages. In cases that there is a direct one-to-one mapping (e.g. UML-C++), there is no need for complex algorithms like ours. However, for two models written in a different language with a complicated mapping and translation, our approach can be applied to synchronization the two models. The extra advantage is that our approach allows the models to be on different abstraction levels, contain more or less details and still provide support for synchronization.

I. 2 Motivation

I 2.1 Why Methodology?

A methodology is a definition of a set of work products and a set of notations, activities and tools structured into a lifecycle process to produce and modify those work products [SPC94]. Each company or even each project group has its own implicit or explicit methodology. The methodology provides the people involved with guidelines when and how to perform certain activities and how different team members should work together. Different methodologies differ a lot in what part of the lifecycle they cover. Each methodology may have elements that are useful to a portion of the development life cycle. The life cycles phases are defined as follows [TOA95]:

- Domain Analysis addresses researching an application domain and identifying, documenting, constructing, testing, and demonstrating reusable components useful in the domain.
- Analysis is that portion of the life-cycle that describes the outwardly observable characteristics of the system, e.g., functionality, performance, and capacity. Normally this description includes models that depict the logical construction of the systems, and its placement within a system environment.
- Design is that portion of the life-cycle that prepares definitions as to how the system will accomplish its requirements. The models prepared in analysis are either refined, or transformed, into design models that depict the real structure of the software product.
- Implementation is that portion of the life-cycle that converts the developed design models into software executable within the system environment. This either involves the hand coding of program units, the automated generation of such code, or the assembly of already built and tested reusable code components from an in-house reusability library.
- Testing focuses on ensuring that each deliverable from each phase conforms to, and addresses the, stated user requirements.

A *complete* methodology is far more than a notation, a process, and some tools. There are organizations that attempt to create fully elaborated methodologies. For instance, Ernst and Young's Navigator method and Andersen Consulting's Foundation method [AC99] consists of thousands of pages bound in a number of binders, provides a number of CD-ROMs, and are coordinated with extensive training. Even “extreme programming” [Beck99], a lightweight methodology that relies on programming in pairs and unit testing during coding, can be considered a full methodology as it provides many organizational and management aspects. In addition to a “notation, process, and tools,” these “complete methodologies” provide [TOA95]:

- Cost Estimating Guidelines,
- Project Management Tasks and Deliverables,
- Measures and Metrics,
- Defined Forms and Deliverable Construction Directions,
- Software Quality Assurance Policies and Procedures,

- Detailed Role Descriptions and Training Programs,
- Completely Worked Examples,
- Training Exercises,
- Techniques for Tailoring the Method, and
- Defined Techniques.

Setting the comments made above aside, this dissertation uses the term "methodology" as consisting of a notation and a process. In this perspective, we provide a methodology that explains the role of UML and SDL during analysis, design and implementation. Testing and domain analysis are only covered partially. A considerable part of our methodology is about notation. UML and SDL require the creation of abstract descriptions and graphical models, of the system under analysis and/or design. These models are constructed using some form of notation. Our methodology specifies which notation should be used for a particular model. The core of our methodology covers the tool support needed for smooth integration of UML and SDL. The different activities in the methodology are brought together in a number of scenarios that describe the process.

The methodology described in this dissertation is *not* a complete methodology and therefore cannot be used directly into real projects. Either our methodology is integrated into an existing methodology or our methodology is extended with those aspects in the list above that are relevant for the project at hand.

I 2.2 Why UML?

It is not difficult to explain why we chose the Unified Modeling Language (UML) [BR95] over other object oriented modeling languages. The UML is the proper successor to the object modeling languages of three previously leading object-oriented methods (Booch, OMT, and OOSE). The UML is the union of these modeling languages and more, since it includes additional expressiveness to handle modeling problems that these methods did not fully address. UML meets the following requirements [OMG99]:

- Formal definition of a common object analysis and design (OA&D) metamodel to represent the semantics of OA&D models, which include static models, behavioral models, usage models, and architectural models.
- IDL specifications for mechanisms for model interchange between OA&D tools. The specification includes a set of IDL interfaces that support dynamic construction and traversal of a user model.
- A human-readable notation for representing OA&D models. The UML notation is an elegant graphic syntax for consistently expressing the UML's rich semantics. Notation is an essential part of OA&D modeling and the UML.

Object oriented modeling languages in general are very useful in our context. They provide a good medium to for specifying, visualizing, constructing, and documenting the artifacts of software systems. The different diagrams each cover another aspect of the system and a set of class diagrams may present the same underlying model in different ways to clarify specific relations. This is also a way to handle complexity, as complex diagrams can be divided into smaller ones. Finally, because OOML in general give little constraints during modeling, is gives the system architect more flexibility to explore solutions.

I 2.3 Why SDL?

Of all existing formal specification languages, SDL [EHS97] is unique in that it combines many qualities: SDL is formally defined by the ITU, it has a graphical notation with one-on-one mapping on the textual notation, SDL specifications are easy to read and understand, SDL's state charts have a high expressive power and a SDL specification can be simulated or transformed to an executable. Because of these qualities, SDL became well accepted by the industry and high-quality professional tools are available [Tele], [Cin]. Section II. 5 gives more details on SDL and provides a mini-tutorial.

Using SDL in a methodology has the great advantage that design, implementation and testing can be performed in the same language. SDL combines powerful structural concepts and expressive state charts and allows object orientation on both levels. Its integration with CCITT [ITU04] and MSC [ITU94-2] allows advanced simulation and testing.

In this dissertation, we limit ourselves to systems that can be specified in SDL. SDL is widely used in the telecommunications field, but it is also now being applied to a diverse number of other areas ranging over aircraft, train control, medical and packaging systems. Some examples for which SDL is not suited are database applications and mathematical libraries. Considering the exponential growth of the telecommunication industry and the circulation of SDL, the constraint of sticking to SDL does not seem to be too restrictive.

The latest version of SDL is SDL-2000, while the research presented in this dissertation is based on SDL'96. The only reason for not using SDL-2000 is that this dissertation reflects the research done before the definition of SDL-2000. However, it was already apparent that SDL-2000 had a strong focus on UML. In fact, many ideas presented in our work, such as linking UML entities to its mapped SDL entity and using the UML extensibility mechanisms, are now standardized in SDL-2000. Furthermore, some complex mappings in our current work are simplified by new features in the language. For example, the composite states of SDL-2000 make it unnecessary to flattening of the UML state diagram before translation. We may conclude that SDL-2000 affirms the idea of UML-SDL round-trip engineering and confirms many ideas presented in this dissertation. More details on the impact of SDL-2000 on our approach are discussed in section VI 3.1 on related work.

I 2.4 Why UML *and* SDL?

We selected UML as the best object oriented analysis language and SDL as the best formal specification language, but why do we need both? Is UML or SDL in itself not enough? Only UML is definitely not enough. None of the UML diagrams is able to specify all details about the behavior of a class necessary to generate executable code. Even if UML could be stretched to make it work [Har97], it is not very practical to write many pieces of Java or SDL into a state diagram. Using only SDL is feasible for smaller projects. SDL could be used as a wide spectrum language from requirements to implementation. But for larger projects, SDL is not flexible enough for analysis and system design. Before one can start specifying in SDL, he or she needs a clear view on the design of the system.

This view of combining UML and SDL is supported by the methodological supplement of the Z.100 recommendation [ITU00]. This supplement recommends the use of OMT for the initial phases of analysis and design and then to pass to SDL. The next version of this methodology guide, which is not yet published, will be updated towards UML, as it is the proper successor to OMT. In addition, SDL-2000 incorporates the most useful UML constructs that were not yet

available in SDL-96 and the traditional graphical representation of UML classes, and Z.109 [ITU99] allows a coherent combination of UML and SDL within the same project.

The main question we answer here is: in which way can UML improve the development or maintenance of a system and how can it be combined with SDL in the best possible way? A good introduction of how UML is used to model real-time systems can be found in [Dou98]. Previous research has been done for combining OMT and SDL in [HWW96].

Our goal is to get the maximum profit of the advantages of both UML and SDL. UML and SDL share a number of qualities, like having a graphical notation, good readability and good tool support. They also incorporate object orientation and state machines, which make UML and SDL suitable to work together. Nevertheless, each of them also has enough advantages to make it worthwhile to combine them both in one methodology. Below we list the most important advantages.

Main advantages of UML over SDL:
- Generic Concepts
- Smooth transition from Use Cases, Conceptual Model and Sequence Diagrams to Class Diagrams and State Charts
- Multiple Views on the same information, i.e. a class can be viewed in several diagrams.
- Little constraints during modeling, more flexibility

Main advantages of SDL over UML:
- Specialized Concepts
- Formal definition and semantics
- Simulatable and executable
- Both graphical and textual syntax

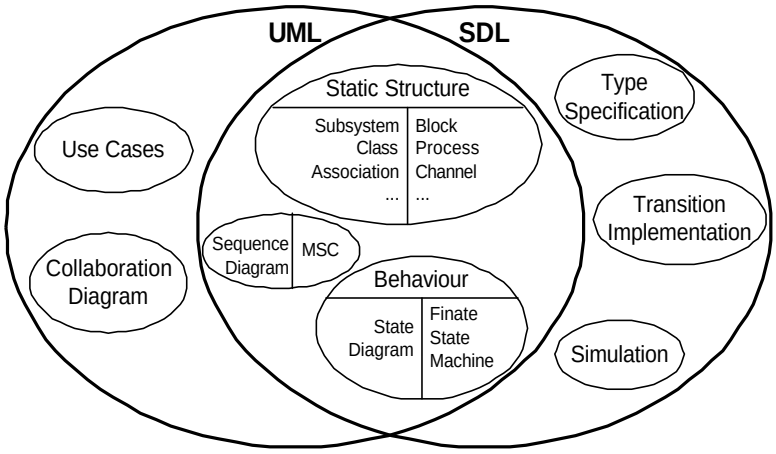

Figure I-1. Comparison of Features of UML and SDL

Comparing the diagrams available in UML and SDL, we come to the same conclusion that UML and SDL is a good alliance. Figure I-1 shows the diagrams or information available in UML and SDL. They share the specification for the static structure, behavior and scenarios. Unique for UML are the use cases and the collaboration diagrams. In SDL the type specifications and the

transitions can be implemented in full detail. Note that UML Sequence Diagrams and MSC's both are used to specify scenarios, but are not dealt with in our round-trip engineering.

I 2.5 Why Round-Trip Engineering?

Knowing that UML and SDL will be used together, one will want tool support for the translation and synchronization of the corresponding information. Without specific support, the UML model is nothing more than a document that can be used as a basis to implement the system in SDL. The developer may put less effort in completing the details of the UML model, as the effort needs to be done a second time anyway. Furthermore, there is no consistency check between the two models and most likely, the UML will not be updated once the development in SDL has started.

The first step in supporting the cooperation between UML and SDL is the one-step translation. The system design phase in UML can be stretched, in order to make a design model that is as complete and detailed as possible. The class diagram and the state diagram of the UML model are then translated completely into SDL. The translator makes extensive use of the UML stereotypes and has a specific interpretation for associations, aggregation and generalization. Technically the translator can translate any UML model, but unless the UML is prepared for the translation, the generated SDL will not be as expected. This preparation is part of the system design phase. The advantages of the translation are apparent. The effort to make a detailed system design model in UML can be reused completely before starting detailed design in SDL. The transition from analysis to design is smoother and many labor-intensive aspects of creating the initial SDL structure and declarations are automated. Still, after the translation, the UML model and the SDL specification are two separate documents. Changes to design should be applied to both UML and SDL. In practice, however, the UML model is likely to become outdated.

The one-step translation can be extended with forward iteration. This means that changes to the UML model are applied to the SDL specification, even if the initial generated SDL specification has already been modified by hand. From a methodological point of view, this is similar to the one-shot translation, but the developer is encouraged to maintain the UML model as it helps him to make design level changes to the system. Provided that all design level changes are applied in UML, forward iteration ensures consistent UML and SDL models. Technically, there are many ways to put forward iteration into practice. Either the updated UML model is translated and then tries to reuse parts of the modified version of the SDL specification. Alternatively, the modified version of the SDL specification is taken as the basis and the changes made in UML are applied to the SDL specification. The first approach is better suited for large changes in the UML model. The second approach is better suited for incremental updates to the UML model. In this thesis, we take the second approach and we extend it with reverse iteration.

The next step toward full round-trip engineering is two-way iteration. Changes in UML are translated to SDL and the other way around. This kind of iteration is applied in discrete steps. Either model is modified and saved, then the iterative translator check for modifications or missing information. It is this kind of support that we present in this dissertation. It allows design decisions to be taken at the best possible side or simple at the side the developer is currently working at.

The ultimate support is the real-time round-trip engineering. It does not matter where or when the design or implementation is done, the other model is always updated automatically in real-time. Tools as TogetherJ [Tog00] show the UML class diagram alongside of the generated Java code in the same window. These tools are based on the one-on-one mapping between UML and Java entities.

I 2.6 Criteria for Evaluation

In this dissertation, we present one possible solution to realize round-trip engineering between UML and SDL. This solution was build with a certain vision of the perfect tool. Here, we present a list of criteria that allows the evaluation of a certain solution in our context. An approach that matches all criteria would be the perfect round-trip solution. Note that our approach is *not* perfect.

Criteria for the evaluation of a certain UML-SDL round-trip solution:

- To boot strap the iteration, the tool should able to translate a complete UML model to SDL or a complete SDL specification to UML.
- The tool should translate and synchronize as much information as possible concerning the static structure of the system, the behavior of individual classes and the scenarios for collaborations.
- The tool should never overwrite or delete manual changes on either side, unless a change in the other side explicitly overwrites it. In particular, the tool should preserve: comments, graphical layout information and analysis or implementation properties of an entity.
- After an iteration, neither model should contain information or constraints that are inconsistent with the linked model, unless the developer explicitly chooses for it. This implies that a developer should have the possibility to indicate that some entity does not take part in the synchronization.
- The tool should allow changes in UML and SDL at the same time.
- In the case of conflicting changes, the tool should let the developer choose interactively between several possibilities.
- The kind of changes that the developer may apply should not be limited to adding code in some predefined area.
- The generated SDL should be readable and easy to change by a human.
- The synchronization should keep working after many iterations, even if the model has completely been changed since the first translation.

I. 3 Overview of the Dissertation

This dissertation in organized in two main parts. Chapters II, II and IV gives the necessary background and overview to understand the technical core presented in Chapter V.

Chapter II positions our research in context of software engineering and provides some background information on object oriented analysis techniques and SDL. The mini SDL tutorial provides the readers that do not know the language with a quick introduction to better understand the rest of the dissertation. Chapter II also gives a summary of the initial research efforts to combine object oriented analysis techniques and formal description languages.

Chapter III gives a comprehensible overview of the mapping of UML and SDL concepts and of the round-trip process. This chapter is particularly interesting, as it gives the reader the necessary background to read and understand the core of this dissertation in chapter V. We also present three scenarios on how the round-trip engineering could be integrated in larger process.

Chapter IV uses an example to illustrates the successive steps in the UML-SDL round-trip engineering. The toffee vendor serves as the example. Starting in UML and an initial translation to SDL, we continue with two iterations through system design in UML and detailed in SDL.

Chapter V contains the technical core of this dissertation: the complete definition of the translation of changes in an UML model to SDL and the other way around. As preparatory work, the information models for UML and SDL are defined, together with the hierarchical links between both models. We define a set of preprocessing rules that prepare a UML model for translation and define how two UML models or two SDL specifications are compared with each other. Sections 6 and 8 of this chapter define the translation of each possible change in respectively UML and SDL. The definition is presented as a large set of translation rules, where each rule has a precondition that states when the rule is applicable or not.

Chapter VI concludes this dissertation by describing the main contributions and discussing related and future research. Noteworthy, we discuss the impact of SDL 2000 on our results.

II. Software Engineering Context

“Always design a thing by considering it in its next larger context -- a chair in a room, a room in a house, a house in an environment, an environment in a city plan.”

-Eliel Saarinen-

“If you don't know where you're going, any road will get you there.”

-Chinese Proverb-

II. 1 Software Engineering

According to the IEEE's definition [IEEE83], software engineering is the systematic approach to the development, operation and maintenance of software in a cost effective way. In our context, we define software engineering as the research for software development methodologies or software processes that allows a systematic approach for the activities above. A software development methodology or software process itself is defined as

- a process for performing particular work tasks throughout the systems development life-cycle and the measures to know they are being done properly;
- a set of tools, methods and notations and a description when and how to use them throughout the life-cycle; and
- the total set of policies, standards, and procedures related to performing software development work tasks.

The aspects covered by a specific methodology can range from only one aspect (e.g. the life-cycle in waterfall model) to a full coverage of all possible aspects (e.g. Andersen Consulting's Foundation method [AC99]). Some additional aspects can be considered from a management perspective; the software development methodology may deal with financial, strategic, commercial and human aspects. We do not cover this perspective in this dissertation.

Having a software development methodology is necessary to cope with the complexity of a software system. To better understand the problems with software engineering, it is interesting to explain how it is different from other engineering disciplines for a number of reasons.

- The process of proving the correctness of software is extremely difficult if not impossible even for small software projects.
- Software engineering deals with abstractions with no physical form (the software). Thus, it is not constrained by materials governed by physical laws or by manufacturing processes as in other engineering disciplines. It is not tangible.
- The software is usually large and complex, thus requires a team or teams of engineers.
- Unlike other engineering products, software usually evolves and requires a great deal of maintenance.

Throughout the short history of software engineering, many methodologies were developed to overcome these difficulties. Most of them improved existing methodologies, but some of them also take radical new approaches. It is clear now that there is no one best methodology for all problems. Instead, a methodology must be selected and tailored for each company or even for each project.

In this chapter, we present some context on each of the three ingredients of a methodology: the life cycle, tools and notation.

II. 2 Life-cycles

In [IEEE83], a software life cycle is defined as "The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life-cycle typically includes a requirements phase, design phase, implementation phase, test phase, installation and check-out phase, operation and maintenance phase, and sometimes, retirement phase." This definition concerns the software life-cycle of an actual system. Below, we define a software life-cycle as the abstract description of how an actual life-cycle should take place. We discuss a number of state of art software processes and their suitability for integrating the UML-SDL round-trip engineering.

One of the first and successful approaches in software engineering is probably the conventional "waterfall" software development life-cycle model as outlined in [Boe76]. This article was based on the original version that appeared in [Roy70]. The model in its pure form received a lot of critique over the years. Most critique stress particular limitations and propose an extension to the original version. Almost every more elaborated software development model has a connection with the principles of the waterfall model somewhere. Just because of this, it is worth mentioning it. Figure II-1 gives an idea of the model. The full model is explained in [Boe81].

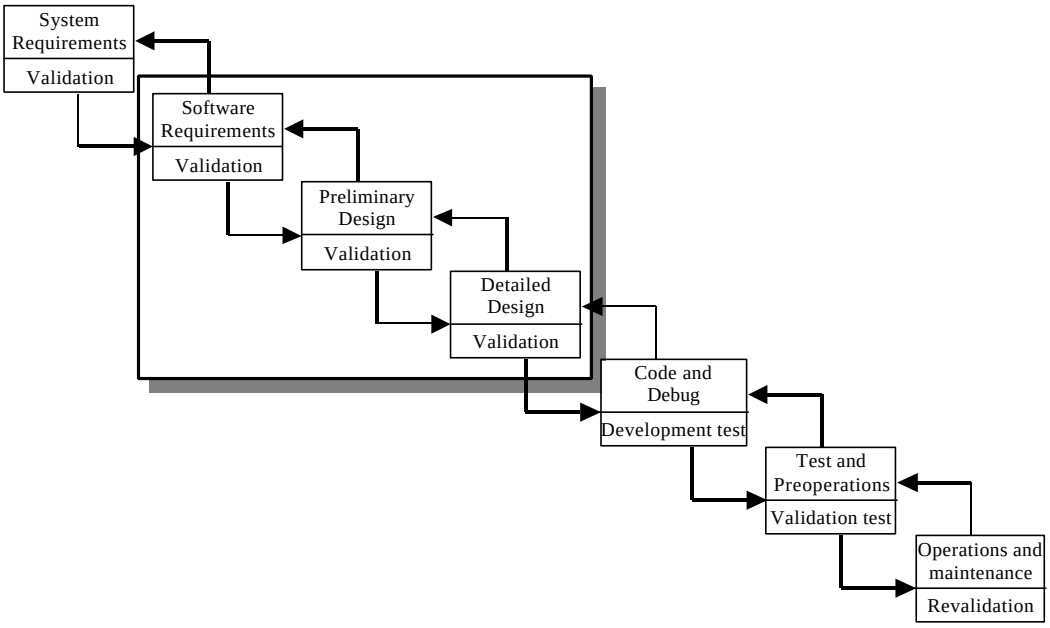

Figure II-1. The waterfall software development life-cycle model

The original waterfall software development life-cycle has seven stages, as shown in Figure II-1. Each of these steps has some build-in validation procedures. These validation procedures are to be fulfilled before the transition to the next step can be done. Failing for the validation procedure can cause a revision of the current step as well as restart at one of the former steps. The validation principle provides feedback to previous steps. After the last step, a global evaluation can be done which results in new system requirements. This can be the starting point for a new project.

Since the publication of the waterfall model, many new paradigms were proposed. Not all of them take the same seven steps. However, the basic principles behind the life-cycle model seem to appear in every methodology. They all have at least the three following steps in the same order: requirements specification, design and coding.

A restricted list of current state of the art software processes include (in alphabetical ordering) Catalysis, Dynamic System Development Method (DSDM), OO software process (OOSP), Object-oriented Process, Environment and Notation (OPEN) and Rational Unified Process (RUP). For each of these processes, we give a short synopsis and discuss their suitability to integrate the UML-SDL round-trip engineering. An in-depth overview of software processes and how to tailor them can be found in [Hig00].

Catalysis [DW98] is a next generation approach for the systematic business-driven development of component-based systems, based on the industry standard Unified Modeling Language (UML). Catalysis provides a systematic process for the construction of precise models starting from requirements, for maintaining those models, for re-factoring them and extracting patterns, and for reverse-engineering from detailed description to abstract models. The ultimate goal is to support the modeling and construction of open distributed systems, i.e. systems whose form and function evolves over time, as components and services are added and removed from it. All work done in Catalysis can be based on composition of existing components, at the level of code, design patterns and architectures, and even requirements specification.

The UML-SDL round-trip engineering fits very well in the Catalysis process. The requirements, analysis and design are based on UML, while the coding language is open. SDL is suitable to specify the kind of systems targeted with Catalysis. Moreover, the process is model oriented and holds in itself forward and reverse engineering steps, making the UML-SDL support very effective. On the down side, SDL is not very component friendly and the round-trip engineering does not give explicit support for building components. We partially solved this issue by defining a component framework for SDL [VWW01].

The Dynamic Systems Development Method (**DSDM**) [Sta97] is a framework of controls for the development of IT systems to tight timescales. DSDM provides a generic process that must be tailored for use in a particular organization dependent on the business and technical constraints. It is independent of any particular set of tools and techniques and can be used with object-oriented and structured analysis and design approaches. The lifecycle that DSDM uses is iterative and incremental. DSDM is particularly well-suited to business applications, where the functionality contains user interfaces (screens, reports, etc.) so the prototyping can be used to maximum benefit.

As DSDM is a generic iterative process, it could be tailored with the UML-SDL round-trip engineering. However, this process targets the wrong kind of systems and requires the involvement of the end-user. Therefore, DSDM is not a suitable process to development of SDL based systems.

The object-oriented software process (**OOSP**) is a collection of process patterns that target medium to large-size organizations that need to develop software that support their main line of business. Similar to design patterns, process patterns describe strategies that software professionals employ to solve problems that recur across organizations. A process pattern describes a collection of general techniques, actions, and/or tasks for developing object-oriented software. The OOSP provides a framework that addresses issues such as how to successfully

deliver large applications using object technology and how to develop applications that are easy to maintain and enhance.

Most of the OOSP process patterns are also applicable when using SDL as a programming language. Still, some additional process patterns for UML-SDL round-trip engineering specific issues should be developed for a full integration with OOSP. The details of using UML during analysis and design should be taken from another methodology, e.g. RUP.

Object-oriented Process, Environment, and Notation (**OPEN**) [GHY97] is a full lifecycle, process-focussed, methodological approach that was designed for the development of software intensive applications, particularly object-oriented and component-based developments. OPEN is defined as a process framework, known as the OPF (OPEN Process Framework). This is a process metamodel from which can be generated an organizationally-specific process (instance). Each of these process instances is created by choosing specific activities, tasks and techniques and specific configurations thereof. OPEN provides strong support for the full lifecycle of a software application.

The OPEN process forms a good framework to fit in the UML-SDL round-trip engineering. Besides the original OML notation, OPEN also supports the UML notation. The management and human relations issues of this process forms a good complement to our models only methodology. Moreover, the OPEN process can be tailored to suit individual domains or projects.

The Rational Unified Process (**RUP**) [JBR99], [Kru99] is a Software Engineering Process built around UML. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. RUP recognizes that no single process is suitable for all software development by making the process configurable. The Unified Process is founded on a simple and

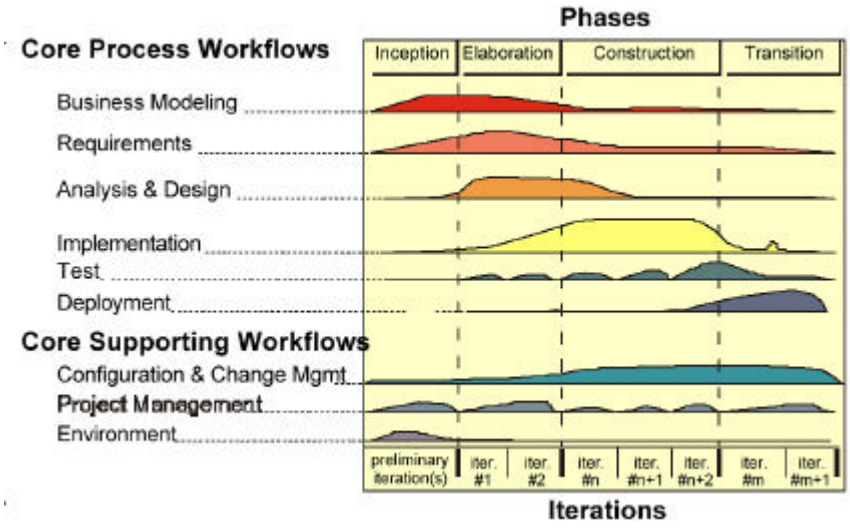

Figure II-2. The Iterative Model of RUP is structured along two dimensions

clear process architecture that provides commonality across a family of processes. Yet, it can be varied to accommodate different situations. The life-cycle model of RUP is structured along two axes. The time axis shows the dynamic aspect of the process and it is expressed in terms of cycles, phases, iterations, and milestones. The content axis represents the static aspect of the process: how it is described in terms of activities, artifacts, workers and workflows. Because RUP is the best fit for our purpose, we included an overview of the content activities and phases in Figure II-2.

The Rational Unified Process is especially well suited for an integration with UML-SDL round-trip engineering. Most importantly, RUP has a strong focus on UML, giving many guidelines for modeling requirements and analysis in UML. Furthermore, the process is fully iterative, making the round-trip support very valuable. RUP appeals to model software visually and to control changes in the software. SDL is a graphical implementation language and our round-trip engineering solution is based on detecting changes. All this makes RUP a good partner for our research.

II. 3 INSYDE Methodology

The INSYDE project [INS94] took place from 1994 until 1996 and laid the foundation to combine object oriented analysis techniques with formal specifications. It was an EU ESPRIT-III funded project. The consortium consisted of Alcatel Bell Telephone (Belgium), Dublin City University (Ireland), Humbolt Universität zu Berlin (Germany), Intracom S.A. (Greece), Verilog S.A. (France) and Vrije Universiteit Brussel (Belgium).

The INSYDE methodology [HWW96] was developed during the INSYDE project and consists of a set of techniques and tools to enable the evolving co-design of hybrid systems from requirements analysis to implementation [SCVM95]. A hybrid system is one that contains significant hardware and software components. The methodology integrates the object-oriented analysis methodology OMT [RBP91] with two domain specific formal description techniques, namely SDL ('88 and '92) [EHS97] for the software side and VHDL [Nav93] for the hardware side. OMT is used as the system requirements analysis technique, and as the technique for the initial system specification. It offers a unified framework for the specification of many application domains in a consistent representation notation throughout the initial design stages. This allows the methodology to provide mechanisms for combining the individual design techniques (OMT, SDL, VHDL), maintaining the consistency of partial models at the detailed design stage and co-simulating the formal description techniques to validate the hybrid system against the system specification. The relative strengths of each design technique (SDL for asynchronous communication systems, VHDL for synchronous reactive systems) can thus be exploited in an optimal way.

Limiting our scope to the transition of OMT to SDL, the functional model of OMT is not useful, so only the static and dynamic models are used. While OMT is a good analysis methodology covering many aspects of system design, the informal nature of OMT makes an automatic translation infeasible. In our methodology, the analysis document is prepared for translation during system design. During this phase subsystems are identified, communication is formalized and information is ordered. To describe these aspects a new language is needed. OMT* [WWV95] is a dialect of OMT, specifically aimed to meet the requirements of system design. OMT* differs from OMT in that:

- It contains only those OMT concepts suited for system design (e.g. no ternary associations, no overlapping subclasses and no hanging classes).
- The possible interpretations of an OMT construct are reduced and clearly described.
- It has a well-defined syntax.
- There are clear relationships between the different models.

In order to make the transition from OMT to OMT* as smooth as possible, OMT* contains as much as possible of OMT and has a semantics that is as close as possible to OMT. Therefore, the semantics of OMT* cannot be defined unambiguously, i.e. the possible interpretations of an OMT* construct is reduced with regard to OMT but not to one single interpretation. Furthermore OMT* does not contain any new constructs and is completely compatible with Rumbaugh [RBP91].

Furthermore, the transformation of OMT* to SDL is assured by giving the language a *transformational semantics* [VJW96], this is a semantics defined by specifying the transformations of a construct to SDL. For those constructs having more than one interpretation, multiple transformations will be specified.

Many concept of the INSYDE methodology are still applicable in the context of UML-SDL'96 round-trip engineering. A UML analysis model should first be prepared during system design before it is ready for translation. The dynamic model of OMT is almost identical to the state diagrams of UML. Consequently, the state diagram translation, including flattening of hierarchical states, can be reused in the new context. Similar to OMT*, the possible interpretations of an UML construct are reduced for the purpose of the translation, but there are still several possibilities left. However, we do not create an UML* as a separate language. We rather define a set of guidelines and preprocessing rules that prepare the model.

II. 4 Object Oriented Analysis & Design

Object-oriented analysis (OOA) is concerned with developing software engineering requirements and specifications of the system's object model (which is composed of a population of interacting objects), as opposed to the traditional data or functional views of systems. Object-oriented design (OOD) is concerned with developing an object-oriented model of a software system to implement the identified requirements. The use of OOD technology requires the development of object requirements using OOA techniques, and CASE tools to support both the drawing of objects and the description of the relationships between objects. Therefore, OOA&D is usually carried out using the same method or language to allow a smooth transition.

Applying OOA&D can yield a number of benefits [Bau96]:

- A better maintainability through simplified mapping to the problem domain, which provides for less analysis effort, less complexity in system design, and easier verification by the user.
- The possibility to reuse the design artifacts, which saves time and costs.
- Productivity gains through direct mapping to features of Object-Oriented Programming Languages.

Many OOD methods have been described since the late 1980s. The most popular OOA&D methods include Booch, Rumbaugh OMT (Object Modeling Technique), Jacobson Objectory, Coad/Yourdon and Shlaer-Mellor. Since late 1990s, however, UML (Unified Modeling Language) is emerging as the defacto standard for OOA&D. For the analysis and design of real-time systems, UML-RT (UML Real Time) has some more problems for getting established.

The UML and standard notation has the formal support of the Object Management Group (OMG) and its various member companies. It's important to realize, however, that the UML is only a standard notation. Essentially, it defines a number of diagrams that you can draw to describe a system, and describes what these diagrams mean. It does not prescribe the process to use to go about building software. Such a process description, or method, would include a list of tasks that need to be done, what order they should be done in, the deliverables produced, the kinds of skills required for each task etc. The original methodologies consist of both notations and a method.

The idea is that by standardizing on the notation, software developers can better communicate providing all the deliverables in a method will use the UML. However, different groups are free to use whichever method they want to use to actually go about building software. Several methods have been proposed that use the UML. Rational has published its Unified Process [JBR99], strongly based on the work of Ivar Jacobson [Jac94]. HP's Fusion [CAB94] method is another method that is widely talked about.

The UML contains so many different diagrams that one needs to decide which are appropriate for the problem a hand.

- Use Case Diagrams

- Static Structure Diagrams: Object Diagrams, Class Diagrams
- Interaction Diagrams: Sequence Diagrams, Collaboration Diagrams
- State Diagrams
- Activity Diagrams
- Implementation Diagrams: Component Diagrams, Deployment Diagrams

In the context of UML-SDL round-trip engineering, we only use the class diagram and state diagram. However, when used in a complete process, other diagrams are used throughout the development process. During analysis, use case diagrams and sequence diagrams are important. During design, sequence diagrams and collaboration diagrams help finding the communication between classes and form a convenient starting point to model state diagrams.

II. 5 SDL as a Formal Specification Language

This section gives a short introduction on SDL as a Formal Specification Language. After some general background information, we present a mini tutorial for SDL. This tutorial explains the minimum that the readers should know about SDL to understand the rest of the dissertation.

SDL (Specification and Description Language) is a standard language for the specification and description of systems. It has been standardized as ITU (International Telecommunication Union) Recommendation Z.100. SDL is a general-purpose description language for communicating systems. Although SDL is widely used in the telecommunications field, it is also now being applied to a diverse number of other areas ranging over aircraft, train control, medical and packaging systems. The key features of the language are:

- suitability for real-time, stimulus-response systems;
- presentation in a graphical form;
- a model based on communicating processes (extended finite state machines)
- object oriented description of SDL components;
- the ability to be used as a wide spectrum language from requirements to implementation.

The language has been evolving since the first Z.100 recommendation in 1980 with updates in 1984, 1988, 1992, 1996 and 1999. Object Oriented features were included in the language in 1992. This was extended in the latest version (SDL-2000) to give better support for object modeling and for code generation. Today SDL is a complete language in all senses. As this dissertation reflects the research done until the end of 1999, our round-trip engineering is based on SDL'96. The impact of SDL-2000 on the round-trip engineering, which is positive, is discussed in section VI 3.1.

II 5.1 Benefits of a Specification Language

It is widely accepted that the key to successfully developing a system is to produce a thorough system specification and design. This task requires a suitable specification language, satisfying the following needs:

- a well-defined set of concepts
- unambiguous, clear, precise, and concise specifications
- a thorough and accurate basis for analyzing specifications
- a basis for determining whether or not an implementation conforms to the specifications
- a basis for determining the consistency of specifications
- computer support for generating applications without the need for the traditional coding phase

SDL has been defined to meet these demands. It is a graphical specification language that is both formal and object-oriented. The language is able to describe the structure, behavior, and data of real-time and distributed communicating systems with a mathematical rigor that eliminates

ambiguities and guarantees system integrity. It has a graphic syntax that is extremely intuitive. Even an SDL layman can quickly obtain an overview of a system's structure and behavior. The most important characteristic of SDL is its formality. The semantics behind each symbol and concept are precisely defined. Above all, the great strength of SDL lies in describing large real-time systems [BH93].

II 5.2 Mini Tutorial

In this section, we present a short tutorial for SDL. We discuss only the most common SDL constructs. Together with the example in chapter IV, this should give enough information for the readers to understand the technical work in de rest of the dissertation. We distinguish four concepts of importance in SDL in more detail: architecture, behavior, communication and data.

II.5.2.1 Architecture

The architecture describes the static structure of a system. The *system* is the highest level in the structure. Everything outside the system level belongs to the environment. The *blocks* are used to partition the system into smaller parts. In this way, the readability of the specification is increased, especially when large systems are specified. The *package* is another structural feature that contains declarations and definitions similar to a system, but does not have its own scope. The contents of a package can be imported by a system or another package.

A block must contain either one or more blocks or one or more processes. Blocks and processes must not be mixed in one block. A hierarchical structure is created with blocks in blocks. Processes describe the behavior of the system.

SDL contains language concepts covering the four basic concepts of object orientation (identity, classification, polymorphism and inheritance). What in traditional object orientation is called a class is in SDL called a *type*, and objects are in SDL called *instances*. Systems, blocks and processes can all be classified in types: *system type*, *block type* and *process type*. Type specifications do not follow the conventional scoping rules, e.g. a process type can be defined on system level and process types and bock types can be mixed within the same scope. A type specification has to be instantiated before it can be used. Typically, one or more block or process specifications are placed in a package or a system level and instantiated in a system. Figure II-3 illustrates the use of the most common SDL structures and structured types.

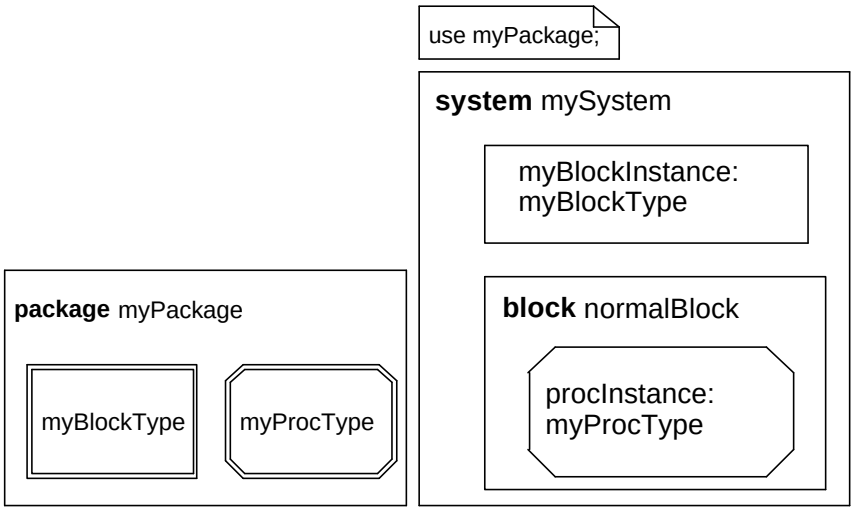

Figure II-3. SDL structures and structure types

II.5.2.2 Behavior

Dynamic behavior is described by processes. Processes execute in parallel and are independent of each other. This means that the status in one process is not known by the other processes in the system.

Processes are defined in the static specification. During run-time when a system is executing, instances from that definition are created. More than one instance of a process can exists at the same time during run-time. Process instances can be created at system startup or created and terminated dynamically at run-time. In Figure II-4, "init" is the initial number of instances at startup and "max" is the maximum number of instances that can exist at the same time during execution.

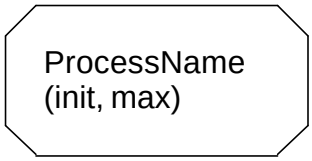

Figure II-4. Process Definition

The model used to describe behavior in processes is the finite state machine (FSM). An FSM consists of states, the example shown at the left hand side of Figure II-5 has two states A and B. Going from one state to another is called a transition. A transition between two states is made after a stimulus has been received. In the example, the state machine is waiting in state A and when *stim1* is received, the transition to B is made. During the transition a number of actions are performed, in the example a response *reps1* is send and *a* is assigned a value. Now the state machine is waiting in state B. If no stimulus is received, then the state machine is inactive, waiting in a state. When a finite state machine is executed, the initial state must be known. Therefore, each FSM contains a start transition, which must not contain a stimulus. The start transition is fired automatically when the FSM is executed.

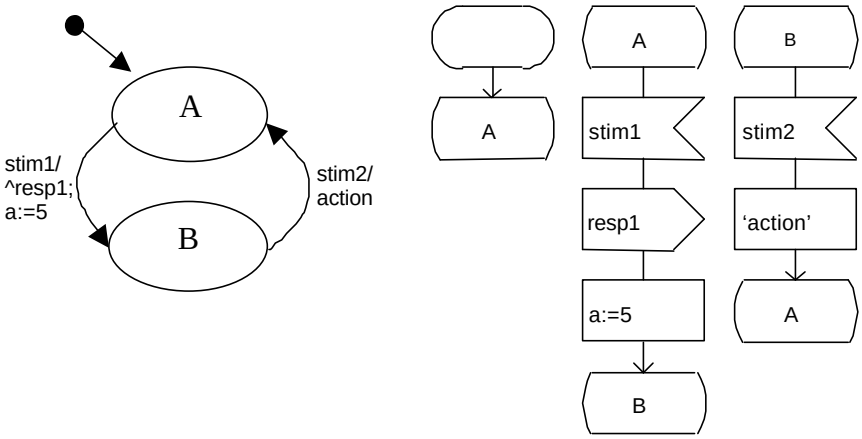

Figure II-5. Example FSM and the equivalent in SDL

A process is a finite state machine extended with data and communication. Figure II-5 shows an FSM and the equivalent state machine in SDL notation. The process consists of two states A and B. State A is defined as the start state. The stimuli in SDL are called signals. The only way to leave a state is to receive a signal, also called an input signal. When the signal is received, the transition is initiated. During the transition, actions can be executed. In the first transition of the example, a signal is sent out. The next state defines the end of the transition and which state to enter next. Most of the time, an SDL process is waiting in a state.

The body of the transitions can contain a wide range of actions. The graphical counterparts of each of the constructs are shown in Figure II-6. The *decision* construct is used to split a transition into two or more branches depending on some condition. *Outputs* are used to send signals and possibly contained values as parameters to other processes, thereby providing a mechanism of communication between processes. Additional information on the destination of the signal can be specified: *to* for a specific destination and *via* for sending the signal through a gate or signal route. A *task* is used to assign a new value to a variable. The *call* construct is used to call a procedure with an optional list of actual parameters. The *create* construct allows a process instance to create another process instance in the same block. The *stop* construct terminates the process instance, thereby freeing its variables from memory. With the *comment* structure, a textual documentation can be added and attached to a symbol.

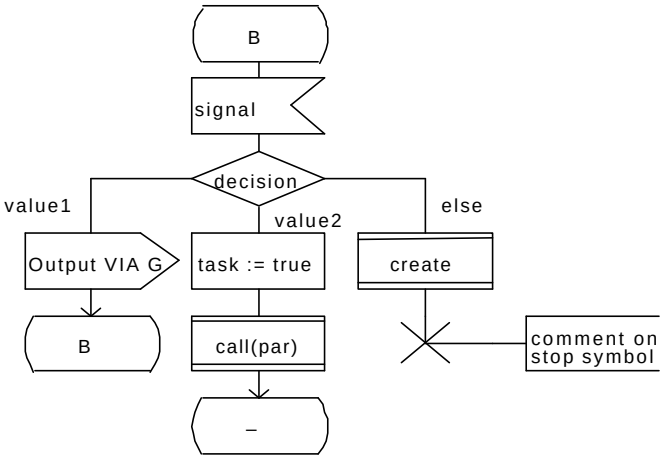

Figure II-6. Transition showing the basic behavioral features

II.5.2.3 Communication

Processes communicate with discrete signals. Communication in SDL is asynchronous, which means that the sending process continues executing without waiting for an acknowledgment from the receiving process.

Signals are defined in a text symbol, as shown in Figure II-7. A signal defined at system level can be used in the whole system. If this is not necessary, the signals are better defined in the block where they are used. The signals can then be used by blocks and processes contained in the structure and by the block itself.

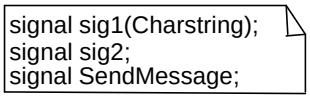

Figure II-7. Text symbols with signals declarations

Channels define the communication path through which blocks communicate with each other or with the environment. Communication with the environment takes place by connecting a channel or signal route with the outer frame of the block. Adjacent to the channel arrow, the signals that can travel on the channel in the arrow direction are stated within square brackets. Signal routes define the communication path through which processes communicate with other or with the block level above. To connect a signal route with a channel, the name of the channel is stated outside the frame; see Figure II-8 for an example.

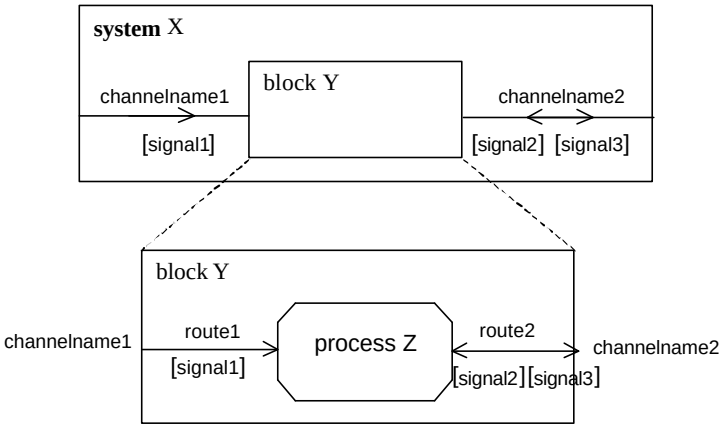

Figure II-8. Connecting Channel and Signal Routes

II.5.2.4 Data

A process can use data stored in variables. Variables can only be defined in processes. Variables are declared with the keyword DCL. The exchange of values between processes is performed by means of a parameters passing mechanism, i.e. values are send along with signals.

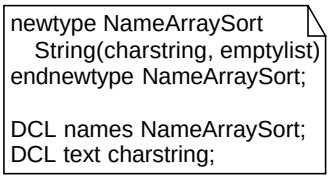

Figure II-9. Newtype and Variable Declaration

SDL provides a powerful method to define data types. The model for SDL data representation is the abstract data type (ADT). Data is represented in terms of values and behavior. The predefined data types are: integer, natural, real, boolean, character, charstring, PId, duration and time. A new data type (called *sort* in SDL) is defined with the NEWTYPE construct, e.g. see Figure II-9. A number of sort generators allow building sorts that are more advanced: structure sort, array sort, enumeration sort, etc. In an enumeration sort, the literals define the values that can be assigned to a variable of that sort.

III. UML-SDL ROUND-TRIP ENGINEERING METHODOLOGY

> “You're either part of the solution or part of the problem.”
>
> - Eldridge Cleaver

III. 1 Introduction

In this chapter, we look at the round-trip engineering from a methodological point of view. We give an informal overview on the mapping of UML and SDL concepts. We describe the phases in the round-trip process and position it into the whole life-cycle.

UML and SDL share a number of qualities, like having a graphical notation, good readability and good tool support. They also incorporate object orientation and state machines, which make UML and SDL suitable to work together. Nevertheless, each of them also has enough advantages to make it worthwhile to use them both in one methodology. Section III. 2 provides a comprehensible description of how UML and SDL concepts relate to each other. The translation rules defined in chapter V give an exact definition of how a UML model is translated, but it is difficult to get a picture of the translation by reading them. The mappings given below are consistent with these definitions and are easy to read and understand.

Section III. 3 gives an overview of the subsequent phases in the round-trip engineering process when performing several iterations. The full example in chapter IV is developed by following exactly this process. The process and mapping description together, provides the necessary background to read and comprehend the formal definition of translation rules in chapter V.

Section III. 4 discusses how the UML-SDL round-trip process fits into the whole life-cycle. The UML-SDL round-trip engineering in itself is not a complete methodology. The notation and process described in this dissertation must be integrated into a larger methodology. In this section, we propose three scenarios of how UML and SDL can be combined: forward engineering, reverse engineering and round-trip engineering.

III. 2 Mapping of UML and SDL Concepts

This section contains a comprehensible overview of the mapping of UML and SDL concepts. It assumes that the reader has reasonably knowledge in UML, as well has a basic understanding of the concepts of the SDL.

The section starts with the mapping of classes and their relationships. Classes with different stereotypes have different mappings. The relationships are associations, aggregations and inheritance, which describes communication structures, hierarchies and inheritance relationships. The last part of the section describes the mapping between UML state charts and SDL process behaviour descriptions.

III 2.1 Mapping of Static Structure

The basic building blocks of a UML model are packages and classes, where the classes represent the active components. Several stereotypes have been defined to give classes various semantics. These stereotypes are «block», «process», «actor» and «newtype». A class in UML without a stereotype is by default transformed into a process; or into a block if the class has a component by aggregation.

In SDL, the basic building blocks of a system are packages, blocks and processes. Blocks and processes also have a typed version, i.e. block type and process type. For this reason, «block» and «process» class can define a property called *typed*. A class with the typed property set to true, is said to be a typed class and maps on a block type or process type. The default value of the typed property depends on the global translation options and on restriction in the model. For example, a class involved in a generalization relationship is always typed.

A class with stereotype «actor» represents an active entity outside the system. In terms of SDL, this corresponds with communication with the environment of the system. A class with stereotype «newtype», finally, represents an abstract data type and maps on an SDL newtype. This gives us the mapping table shown in Table III-1. Figure III-1 shows an abstract UML model with the mapped SDL system. The class *C1* is typed and is mapped on the process type *C1*. The class *User* with stereotype «actor» is mapped only indirectly through the association *as*, which is mapped on the channel that going to the environment.

UML	**SDL**
Model	Specification
Package	Package/System
Non-typed «block» class	Block
Typed «block» class	Block Type
Non-typed «process» class	Process
Typed «process» class	Process Type
«newtype» class	Newtype definition
«actor» class	Environment

Table III-1. Mapping of basic structures

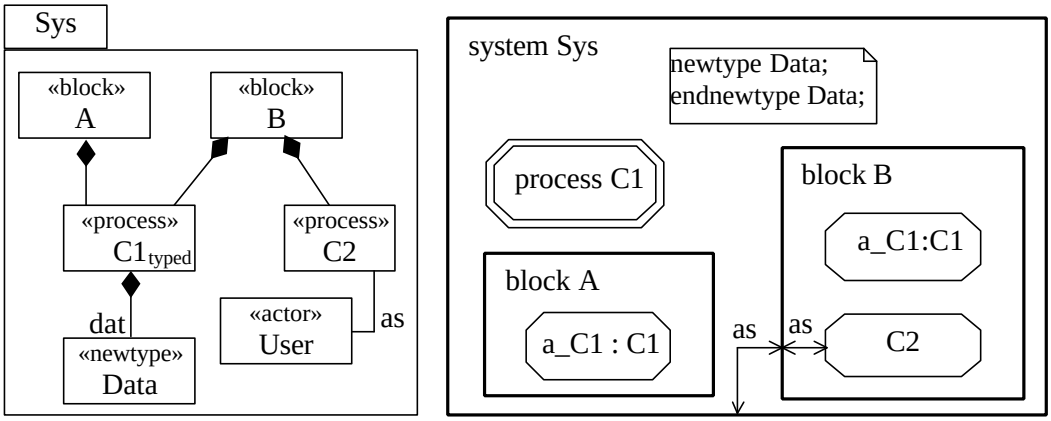

Figure III-1. Example of Structural Mapping

Another aspect of the static structure is the relationship between classes. Basically, associations map on communication in SDL and aggregation maps on nested structure. In fact, things are a bit more complicated than this, because on the UML side the semantics of the aggregations depends on the kind of component and on the SDL side we have different possibilities to allow communication. Table III-2 shows an overview of the mapping of the relationships. In Figure III-2, the typed class *C1* is a component of classes *A* and *B* and therefore. Both aggregations are mapped on a type-based process. In the case of class *C2*, the aggregation makes the mapped process appear in block *B*, but does not have a mapping on its own.

UML Concept	**SDL Mapping**
Aggregation to typed class	Type based instance
Aggregation to non-typed class	Scope of definition
Aggregation to «newtype» class	Variable declaration
Generalization	Inheritance
Association between «process» classes	PId pointer
Association to «process» class	Signal Route and/or Pid variable
Association to «block» class	Channel(s)
Role of association to typed class	Gate
List of Operations	Signal list

Table III-2. Mapping of UML relationships

The mapping of communication needs some elaboration. The basic rule is that for every association between two non-abstract classes, there must be a communication route between the corresponding processes. However, this can be achieved in different ways. For two processes in the same block, there is no problem, the association maps on one single signal route between the two processes. But in order to connect two processes in a different block, we need two partial signal routes (i.e. signal routes to the environment) and a number of channels connecting the two signal routes. In order to maintain readability in larger systems, channels are merged as much as possible. See Figure III-2 for an example of merged channels. The channel *assoc* generated between block A and block B is reused to connect the signal routes assoc3 and assoc4.

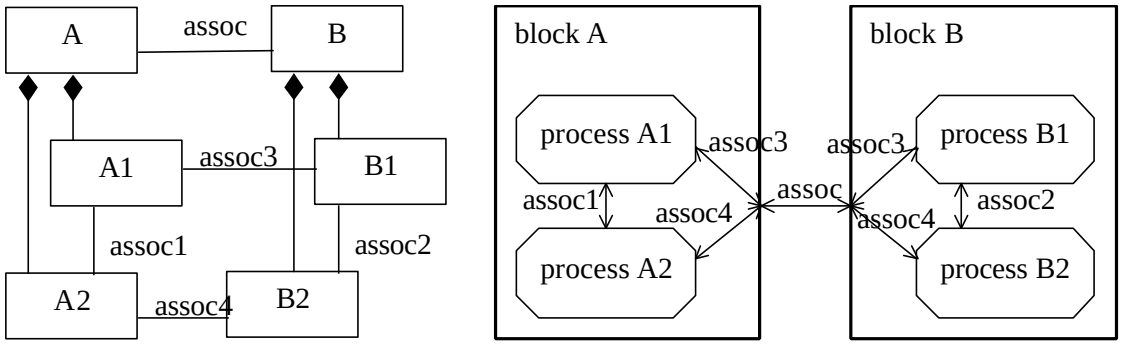

Figure III-2. Translation of Communication

III 2.2 Mapping of Declarations

Other aspects of the mapping are the signal, type and variable declarations. Operations with stereotype «signal» maps on a signal declaration. For each class, a signal list is generated that contains all the signals of the class. Operations with stereotype «procedure» maps on an SDL procedure definition. By default, operations with a return type get the stereotype «procedure» and operations without a return type get the «signal» stereotype. Attributes of active classes map on SDL variable declarations. Table III-3 gives an overview of the declaration mappings.

UML Concept	**SDL Mapping**
«signal» Operation	Signal definition
Set of Operations	Signal list
Attribute • Public	Process Variable • Exported & Remote
«procedure» Operation • Private Operation • Public Operation • NA • Parameter	Procedure Definition • Local Procedure Def. • Exported Procedure Def. • Remote Procedure Def. • Formal Parameter

Table III-3. Mapping of Declarations

III 2.3 Mapping of State charts

The mapping of UML state diagrams on SDL state diagrams is rather straightforward, except for nested state diagrams and entry and exit actions.

UML Concept	**SDL Mapping**
State Diagram	Final State Machine
Initial State	Start
State	State
Final State	Stop
Nested State Diagram	State lists and/or Flattening
Submachine State	Procedure & Procedure Call
Entry/Exit Action	Action on Transition
Outgoing Transition	Transition

Internal Transition	Transition to –
Output Event (Destination)	Output Signal (To Pid)
Action	Action

Table III-4. Mapping of state charts items

Nested State Diagrams

The UML state diagrams include the notion of nested hierarchical states. This concept is inherited from the Harel statecharts [Har87]. Basically it means that a state can contain substates and while being in a substate, the state machine will also fire transitions originating from the superstate. In the current version of SDL, nested states are not available, but SDL has the notion of statelists. A transition can be added to several states at the same time by listing the states in the state symbol.

It can be shown that a hierarchical UML state diagram can be correctly translated to SDL using statelists when none of the substates have exit actions. In the other case, the substates containing exit actions, must be excluded from the statelist and get a duplication of all the transitions. Figure III-3 shows an example of a nested state diagram in UML and SDL. Note that, unlike the exit action, the entry action in state Sub2, does not cause the duplication of transitions.

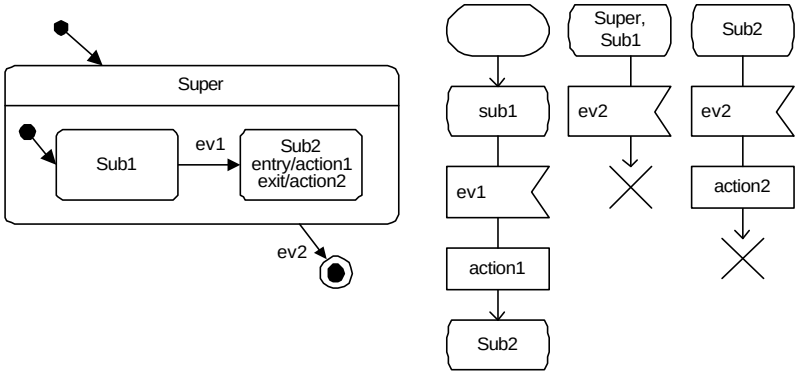

Figure III-3. Flattening a State Diagram with Entry & Exit actions

UML differentiates between six events, each of which has a different mapping as shown in Table III-5. In UML, an event is what triggers a transition. There is always exactly one event for each transition, possibly guarded with a guard-condition. Please refer to the UML semantics definition for a comprehensive explanation of all different events.

UML Event	**SDL Mapping**
Call Event	RPC input
Guard on Event	Enabling Condition
Change Event (e.g. [x < 10])	Continuous Signal
Time Event (e.g. after 5 sec)	Action on Transition
Signal Event	Input
Empty Event (or lambda transition)	Spontaneous Transition
Deferred Event (within state)	Save (Signal / RPC)

Table III-5. Mapping of Events

In UML, an action is anything that happens on a transition. Compared to OMT, UML differentiates between many kinds of actions, which makes it easy to achieve a detailed mapping, see Table III-6.

UML Action	**SDL Mapping**
Send : ^target.event(parameter)	Output event(parameter) TO target
Call	Procedure Call
Create	Create
Terminate	Stop
Uninterpreted • With ":=" • With "call" • Other	SDL Expression • Assignment • Procedure call • TASK ' '

Table III-6. Mapping of Action

A number of UML concepts cannot be mapped to SDL because there is no equivalent and it would be too awkward to translate them. First, there are the history and deep-history states. Translating the history state to SDL would require duplicating the complete state diagram, which is unacceptable. Second, there are the concurrent and non-concurrent composite states. Composite states introduce concurrency within one class, but it is impossible to have concurrency in one SDL process. Related to this the fork and join states are not mapped either. If either of these constructs appear in a state diagram, they are simply ignored. For a full translation, the state diagram must be refactored without using the unsupported constructs.

On the other side, there are a lot of SDL constructs that cannot be expressed in UML. For example the body of a newtype declaration, connection of multiple channels, timers in general and priority signals. There is one SDL feature that is particularly difficult to translate, namely the join. The join, together with the decision, allows one transition to have different paths and merge after some actions. For a correct reverse engineering from SDL back to UML, all actions after the join must be duplicated, resulting in a many on one mapping.

III. 3 Incremental Round-Trip Engineering

In this section, we explain the basic principles of our round-trip engineering approach. It gives a comprehensible view on the incremental round-trip engineering process defined in chapter V. The mapping definition, given in the previous section, forms the basis for a translation from UML to SDL and the other way round and for tool support for synchronizing a UML model and an SDL specification. Because the mapping is not a strict one-on-one mapping, traditional round-trip engineering solutions cannot be used. Instead, we use a set of translation rules that define how changes in UML model are translated into changes in the SDL specification and the other way-around. Some examples of possible changes are: new class, rename operation, delete association, etc. These changes are automatically detected, translated to SDL (or UML) and applied locally on the specification with maximal preservation of detailed design changes in SDL. Hierarchical links between UML and SDL syntactic elements provide the context in the SDL system where to apply changes. The "new class" change, for example, is translated by adding a block or a process to the block that is linked with the aggregate of the new class. Other information available in the parent block is kept untouched, including the graphical layout.

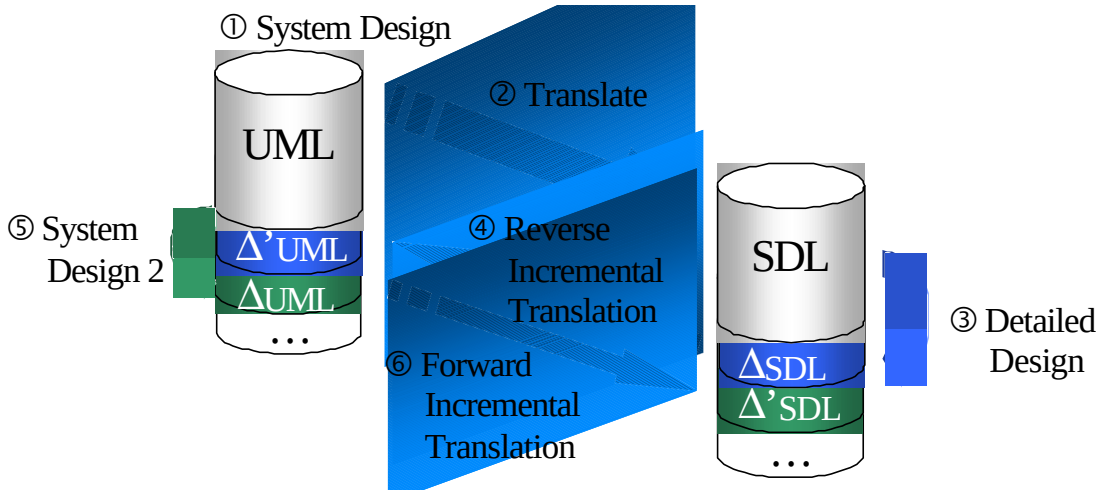

Figure III-4. Successive Iterations

Figure III-4 depicts the first few phases in the round-trip engineering process. It consists of successive executions of forward incremental translation and reverse incremental translation. The round-trip engineering always starts with a more or less complete UML model ①. In other words, it is currently not possible to start the process by reverse engineering an existing SDL specification. The class and state diagrams of UML model are translated into a full SDL specification ②. Because the translation is defined in the form of changes, the first UML model is virtually compared with an empty model, such that all entities in the model are considered "new". During detailed design, the generated SDL specification is further refined ③. Detailed design includes refactoring the generated SDL, as well as adding new functionality. Next, the changes made in SDL are detected by comparing it with the original SDL specification and translated back to UML ④. Similarly, the updated UML model is improved during system design ⑤ after which the changes are detected and translated to SDL ⑥. In order to allow incremenal translation of

changes, the corresponding enties in the UML model and SDL specification are linked with each other.

The algorithm that compares two models looks for new entities, deleted entities and matched entities based on the entities' unqiue identifiers. An entity that is present in the new model but not in the old model is translated as a "new" entity. An entity that is present in the old model but not in the old model is translated as a "deleted" entity, usually by deleting the linked entities on the other side. An entity that is present in both models is said to "match". The attributes of matching entities, such as name and type, are futher compared in the translation rules. In this sence, the translation rules themselves also perform a part of the comparison.

III. 4 Three Scenario's for combining UML and SDL'96

The UML-SDL round-trip engineering process alternates between system design and detailed design. However, these two phases do not make a full life-cycle. The round-trip engineering should be fit into a larger methodology. In this section, we describe three different scenarios how the UML/SDL translation can be embedded:

1. Forward Engineering: This scenario is followed for new projects. The requirements analysis and system design is done in UML. The system design model is then translated to SDL, where the development continues with the round-trip scenario.
2. Reverse Engineering: This scenario is followed in the case that there is already an SDL specification available. The specification is translated to UML, either for documentation purposes or for reengineer purposes. Altough the same mappings can be used, the current set of translation rules do not support the reverse engineering of a full SDL specification.
3. Round-trip Engineering: After either scenario 1 or 2, there is UML and SDL available for the same system. From then on the two models are kept synchronous by forwarding the changes made on the other side.

III 4.1 Forward Engineering

In this first scenario, the developer starts building a new system in UML, which offers many advantages to start building a system from scratch. Figure III-5 shows an overview of the activities in this scenario.

The set of external objects and their interaction with the system form the basis for the requirements analysis of the system [Dou98]. Use-case diagrams and collaboration diagrams are especially suited for this task. These, together with sequence diagrams, allow a smooth transition to system analysis.

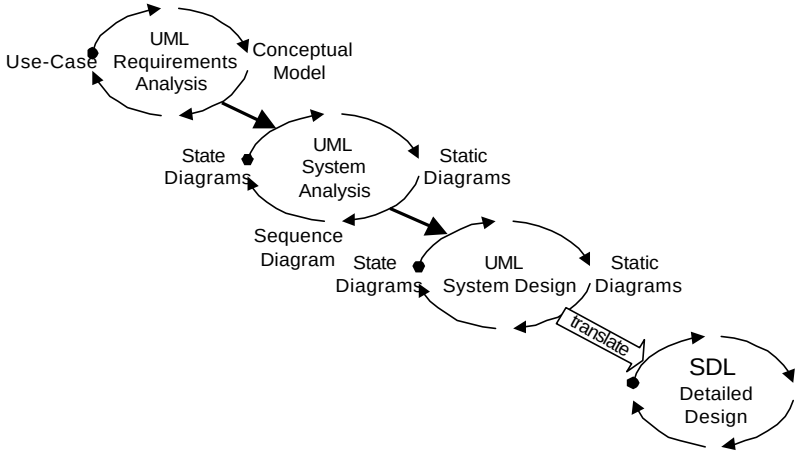

Figure III-5. Forward Engineering Scenario

The goal of system analysis is a static and behavior model of all the important components in the system. Starting from the requirements, the analyst must identify the key objects and classes and their relationships within the system. Using the first set of classes, a sequence diagrams is made for a number of use-cases, possibly detecting missing classes. The classes can also be extended with operations now; each message the class needs to understand becomes an operation.

The last step before the translation to SDL is system design. Still a lot of design decisions need to be taken here. First, the classes need to be grouped into subsystems and packages can be split. All attributes and operation parameters must be given a type and undefined types must be defined. If state charts are used, all concepts that cannot be translated, like history states and concurrency, must be eliminated. It also matches the signals that are sent in any state diagrams with the «signal» operations in other classes.

During detailed design, the developer can continue development based on the generated SDL. Typically this encompasses activities like –but not limited to– filling type declarations, improving channel definitions, completing details on transitions, process initialization, creating timers, adding formal parameters, etc. In other words, detailed design adds everything in SDL that cannot be expressed in UML. Note that this is more than only filling in the details; it is a continuation of the design.

At this stage, we enter the round-trip scenario because we have a UML model and an SDL specification of the same system. From hereof, most changes one side can automatically be reflected on the other side. Please see section III 4.3 for the continuation of the development life cycle.

III 4.2 Reverse Engineering

The primary reason to reverse engineer an SDL system is to get a different, more abstract view on the system. This view can either be used as documentation to give better insights into the system or as a basis to refactor, restructure or reengineer the SDL system. The main advantages of using UML for reverse engineering is that it allows multiple views on the same information and that it incorporates other diagrams that can help documenting the system. The current set of translation rules do not support the reverse engineering of a full SDL specification. Nevertheless, we briefly discuss two possible reverse engineering scenarios. Figure III-6 shows an overview of both scenarios.

III.4.2.1 Reverse Engineering for Documentation

If the UML serves for documentation purposes, the developer chooses some parts of the SDL specification and gets the UML view of that part. He could extend the UML model with collaboration diagrams, sequence diagrams to make his insights explicit and available to other developers. When a large part of the development time is spent on reading the specifications, creating documentation in this way is a big asset.

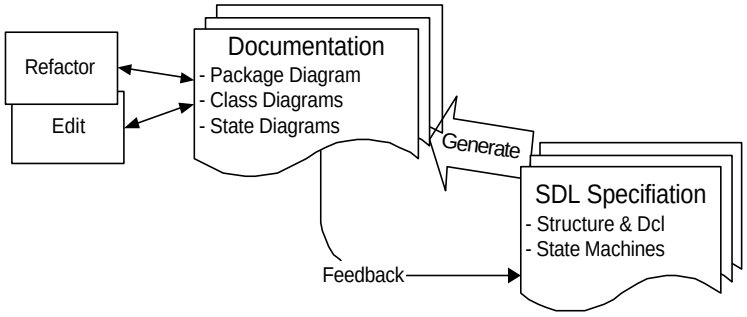

Figure III-6. Reverse Engineering for Documentation

III.4.2.2 Reverse Engineering for Round-trip Engineering

In this case, the developer intends to make modification to the SDL system based on the UML model. The kind of the modifications can vary from behavior preserving refactoring to total restructuring for re-engineering.

For this purpose, the complete SDL specification is reverse-engineered to UML. Although some round-trip support could be given on partial UML models, it would be very error prone because there is no context information about signals, declarations, packages, etc. Therefor the entire SDL specification is translated as complete as possible. The developer then improves the generated diagrams and creates more diagrams for specific views on the system. Now we have a UML model of our system, so we can continue with the round-trip scenario in section III 4.3.

III 4.3 Round-trip Engineering

In this scenario, the developer already has an UML and SDL model of his system and he would like to keep them synchronized. In this way, he profits from the advantages of UML and SDL during the whole development and maintenance life cycle.

The main advantages of UML in a round-trip scenario is that it allows multiple views on the system and thus can give an abstract view on the system as well as the details on certain topics and relationships between classes in different sub-systems. It is also easier to make structural changes, because there are fewer limitations in how you can edit. In addition, the developer can still use all the UML diagrams to extend the requirements or continue the system analysis.

The main issue in round-trip engineering is of course that no matter which model is modified, the common information in both models is kept up to date. Although any change may be made in either model, there is a difference in the typical changes you make in UML or SDL. Figure III-7 shows the interaction between the different activities during round-trip engineering.

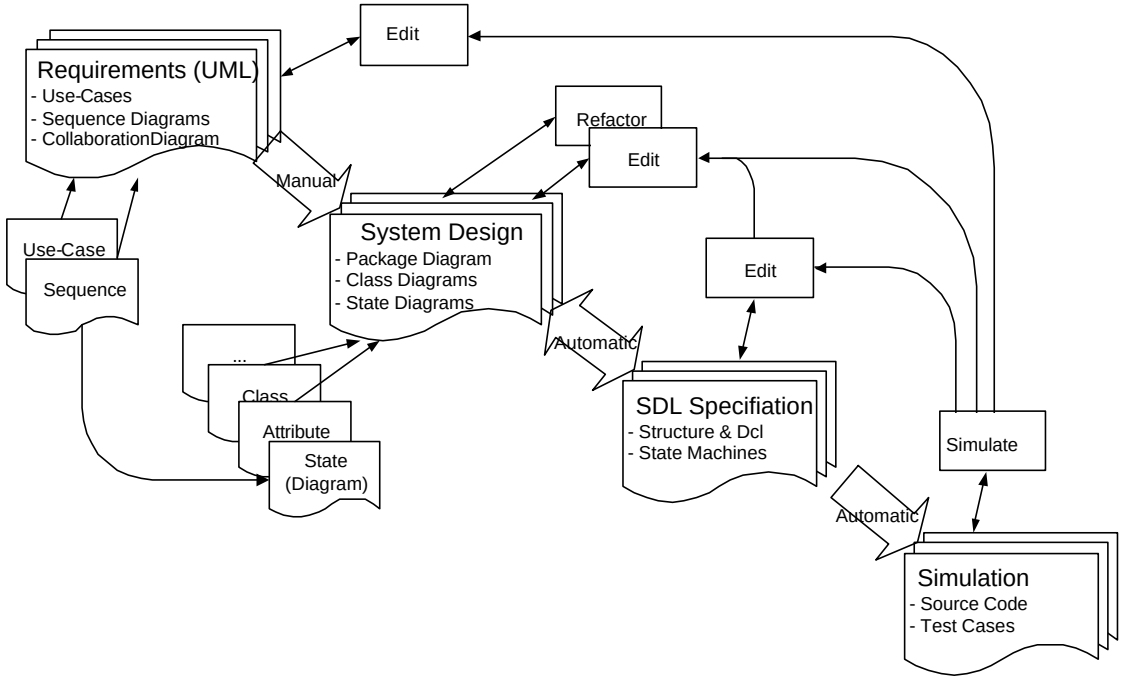

Figure III-7. Information flow during Round-Trip Engineering

UML is better suited for high level changes. For example, for new requirements, you start creating one or more use cases and examine how it influences the class model and state diagram. Alternatively, as the system become bigger, it is probably a good idea to restructure the system on the analysis level. In addition, new design insight might lead you to refactor the class and state diagram, e.g. splitting a class in two classes or applying a design pattern.

SDL is better suited to get all the details right. The first goal is usually to make the SDL specification ready for simulation. First of all, the things that could not be expressed in UML are added, e.g. timers, sorts specification, creation of process and Pid handling. The generated SDL specification can also be optimized from an SDL point of view, e.g. merging channels, moving signal declarations, creating signal lists, etc. Most important, the SDL system can be simulated, giving valuable feedback to the design and implementation. By observing the running system, you will find errors, missing parts or extra requirements, making the circle round.

IV. EXAMPLE

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

-C.A.R. Hoare

"Things turn out best for the people that make the best of the way things turn out."

-John Wooden-

IV. 1 Introduction

This chapter demonstrates the UML-SDL round-trip engineering process by following two iterations in the development of the toffee vendor example (presented in [EHS97]). The main purpose of this chapter is to explain the concepts of this dissertation in an intuitive way. While the extensive list of translation rules of chapter V are hard to read and comprehend, it is easy to understand this concrete example. We start with a UML design model of a simplified version of the toffee vendor and explain the specific use of UML during system design. Then we translate the UML model into an SDL specification as a one-shot translation and explain the different elements of the generated SDL specification and how the generated SDL is linked with the UML model elements. Next, we alternately make improvements to the system in UML and in SDL and show how it affects the other model. The changes are chosen as to illustrate many different features of the round-trip engineering. We show that, in some particular cases, the input of the user is necessary to perform the translation of changes.

The toffee vendor used in our example is taken from [EHS97] and adapted for our needs. In our initial version, the user starts by chooses an item (chocolate, coffee or gum). The system checks whether the requested item is still available. If so, the coin slot is opened and the price is displayed for that particular item. Then the user starts inserting coins. Every time a coin is inserted, the displayed price is updated to reflect the amount due. Once the full price is paid, the requested item is delivered. Figure IV-1 shows the sequence diagram for buying a chocolate. In the initial version, nothing is provided for any exceptional cases; nor does the user get any change back if he pays too much. New features are added in the sections below.

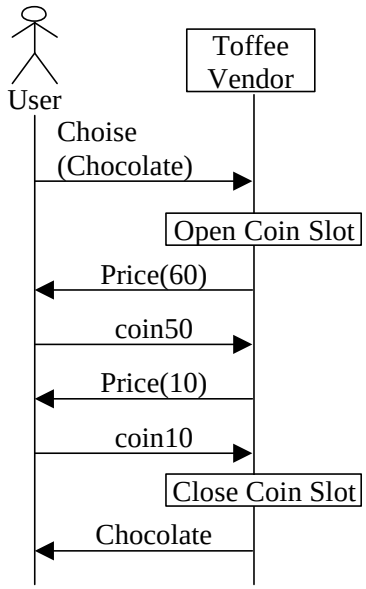

Figure IV-1: Typical use of Toffee Vendor.

IV. 2 System Design

We start the round-trip engineering process of the toffee vendor by presenting the initial system design model in UML. The system design model consists of a class diagram and three state diagrams. System design is in it selve again an iterative process between modeling the class diagram, state diagrams and inspecting the generated SDL code. Here, we start with a first version of the class diagram that does not take the translator specific issues into account. Next, we show the state diagrams for the three active classes. The SDL system generated from this model is not satisfactory; therefore, we revisit the class diagram to fine-tune it toward the translator. Among others, this means that all classes get a stereotype, associations get a name and all attributes and parameters get a type definition.

IV 2.1 Class Diagram

We partition the behavior of the toffee vendor in three classes as shown in Figure IV-2. The *Coins* class accepts the coins from the user and transforms it to an actual value for the controller. The *control* class receives the order and keeps track of the amount due. The *WareMgr* (Ware Manager) checks the availability of a specific article and delivers the ordered item when triggered by the *control* class. The *User* class represents an active entity outside the system. Actors supply stimuli for the system, but can also receive signals.

The operation declarations in the classes are used as signal input declarations. Each class should declare all the signals it can receive as an operation and the parameters of the operation match the parameters of the signal. For example, the *Control* class can receive four signals. The *Choice* signal has one parameter to indicate the *article* the user has chosen. The signals *Empty* and *NonEmpty* are sent by the *Ware Mgr* to indicate whether the requested item is available or not. The *Money* signal, finally, is sent by *Coins* whenever a coin is accepted. The parameter *value* of the *Money* event indicates the value of the coin.

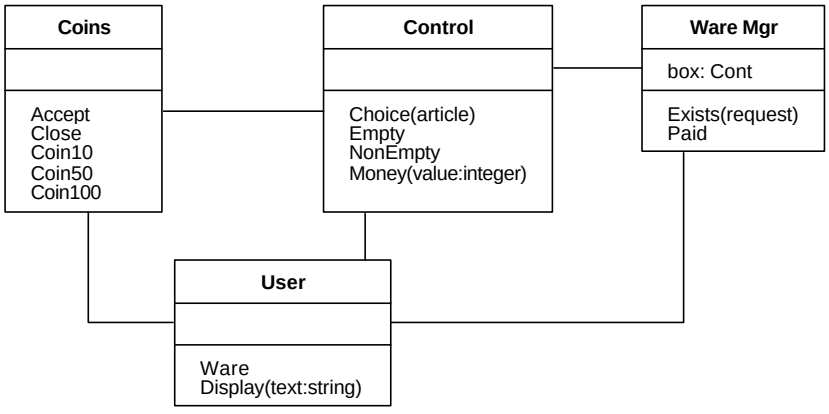

Figure IV-2. Initial Class Diagram of Toffee Vendor

The default semantics of associations between two active classes when translating to SDL is communication. It means that some or all instances of one class are able to exchange signals with

some or all instances of the second class. The specific code generated from an association depends on the translation options. All associations in our example model communication. The associations to the actor *User* denote communication of the system with its environment.

IV 2.2 State Diagrams

We now give some details about state diagrams of the three active classes. A UML class can have one state diagram that describes the behavior of that class. It describes what actions are taken given a certain state and stimulus. In translation to SDL, most of the UML state diagram concepts can be used, including entry and exit actions, output events, initial and terminal states and nested state diagrams. However, many of these features are not available in SDL. Therefore, the UML state diagram is flattened as part of the translation.

Figure IV-3 shows the state diagram of the *Control* class, which demonstrates many special features. Starting in the initial *idle* state, the controller waits for the user to make his *Choice*. It asks the ware manager to check the availability for the requested article by sending an *Exists* signal. The ware manager responds with an *Empty* or *NonEmpty* signal. If the article is not empty, the user can start paying. The *payment* state in has two entry actions. The first entry action is an assignment, recognized by the ':=', and initializes the cost for the article. The second entry action *Accept* is an output event to *Coins*, because *Coins* declares it as an input event. The internal transition *Money* decreases the cost every time a coin is inserted. The transition from *payment* to *idle* is a guarded transition. Whenever the cost becomes zero or less, the ware manager is notified that the article is paid and the control returns to the *idle* state.

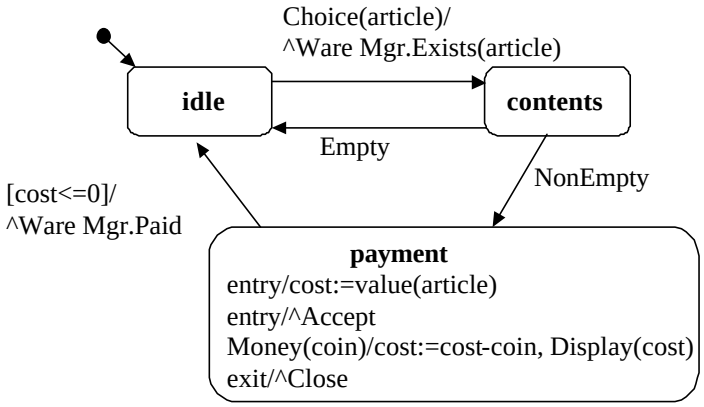

Figure IV-3. State Diagram of the *Control* Class

Figure IV-4 show the state diagrams of the classes *Ware Mgr* and *Coins*. The overall behavior of the state diagrams is easy to understand. The ware manager checks for the availability of an item. If it is available, it waits until it gets a Paid message and then delivers the item. The Coins class accepts different coins and translates them into a value that is manageable by the control class.

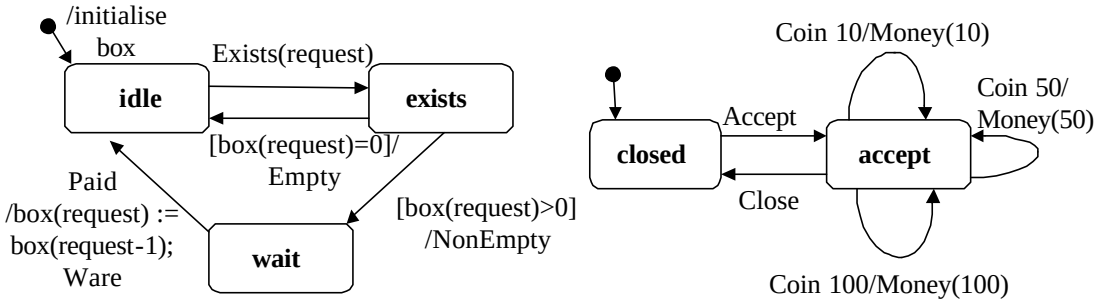

Figure IV-4. State Diagram of the *Ware Manager* and *Coins*

IV 2.3 Investigate Generated SDL

We investigate the SDL system generated from the UML diagrams as we modeled them until now. Figure IV-5 shows the generated system, block and process structures. The system *ToffeeVendor* contains four block instances (single rectangle) and four block types (double rectangle). The *WareMgr*, *Control* and *Coins* block types contain a process with the same name. There are two main problems with this structure. First of all, the *User* class is supposed to represent the environment of the system and therefore does not need a block on its own. Second, it is overkill to generate an extra block and block type for each process, certainly if we consider the burden for extra communication routes.

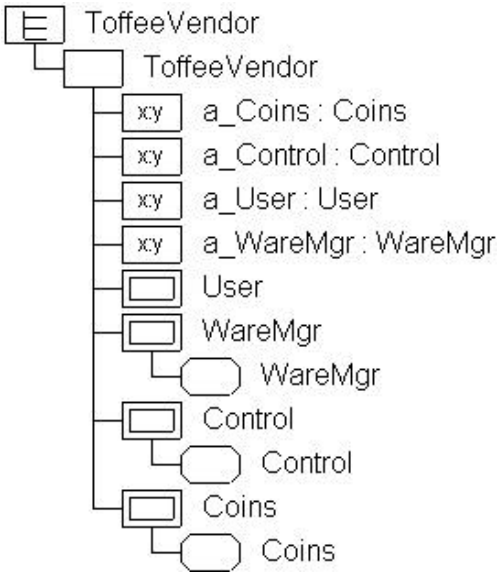

Figure IV-5. Structural overview of the generated SDL System

Figure IV-6 shows a part of the generated signal and type declaration in the ToffeeVendor system. Again, we find a number of undesirable specifications. The parameter of the signals Exists and Choice is *a_request*, while it should state the type of parameter. The reason is that the type is not defined in the class diagram and default translation for a missing parameter or variable type is to prefix “a_” to the name. For the same reason the translator generates two new type declarations

a_request and *a_article*, while they should be the same type. These incorrect types are also used as type for the variable declarations in the processes.

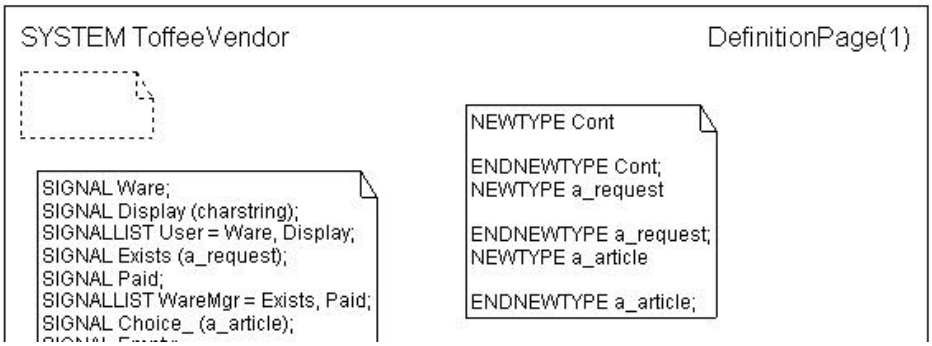

Figure IV-6. Declarations in the ToffeeVendor System

IV 2.4 Class Diagram Revisited

The problems found by investigating the generated SDL could be solved directly in SDL. However, this would require making updates on many different places. After regrouping the processes and deleting the User block and block type, almost all channels and signal routes would need to be rewired. After renaming the new types, all parameters and variables using the type need to be updated.

Before touching the generated SDL, we first improve the UML class diagram. In general, one will continue working on the UML model until the generated SDL looks right at first glance. The round-trip engineering process benefits from a stable starting point. Figure IV-7 shows the improved class diagram of the toffee vendor example. First of all, we fill in the stereotypes of all classes. The stereotype of a class is important information for the translator. The *Coins* and *Control* classes get the stereotype «process» and are grouped together in the «block» *Dialogue*. The process for ware manager is kept separate. The *User* class has the stereotype «actor», which means that it represent an active entity outside the system. The two new classes *Item* and *Cont* (from Contents) are abstract data structures, indicated by the «newtype» stereotype. They are used as the type for attributes and parameters in *Control* and *Ware Mgr*. In our model, *Item* is an enumerated type to represents the choice made by the user. *Cont* is dictionary table that stores the amount of items available for each kind of item. We will fill in the exact details for this type in SDL, because SDL provides the specific constructs to specify ADT's. Some extra attributes that are used during the execution of the state diagram are added to *Control* and *Ware Mgr*. For example the *article* attribute of the *Control* class shown in Figure IV-7 is used to store the user's request.

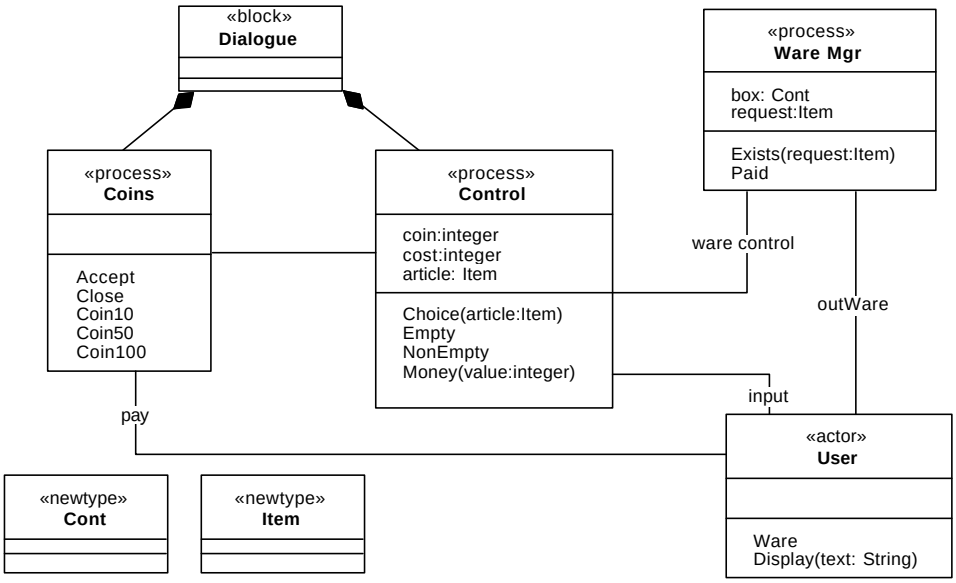

Figure IV-7. Improved Class Diagram of Toffee Vendor

IV. 3 Translating UML to SDL

The first step in iterating between UML and SDL is the generation of an SDL specification as a base for detailed design. In order to use the incremental translation rules, the UML model is virtually compared with an empty UML model, i.e. all classes, attributes, states, etc. are all translated as new entities. These translation rules are designed to generated readable SDL specifications that are a good base for detailed design.

The translation is adjustable by a number of parameters, depending on the purpose of the generated SDL. For the toffee vendor example, we choose to generate a system (as opposed to a package), as the application will run standalone and is not a part of a bigger system. We choose to generate block types and process types to have more flexibility for future extensions. We choose to generate signals globally because the system is rather small. Finally, we choose not to generate PId variables, as no processes are created at run-time.

Here we present an outline of the translation rules and apply them to the Toffee Vendor example. Before the actual translation, the UML model is preprocessed to check for missing information or inconsistencies. We divide the translation itself into three parts. First, we generate the structural objects of the SDL description. Then, the processes are interconnected with communication routes and declarations are generated. Finally, a finite state machine is generated for each process.

IV 3.1 UML Preprocessing

To assure a correct execution of the translation rules, the UML model must first be preprocessed. Although the class diagram of our example is quite complete already, a number of things need to be done here. The *Ware Mgr* class has the stereotype «process», but it does not have an aggregate. As this is not allowed, its stereotype is changed to «block». The association between *Coins* and *Control* does not have a name. As a default, the names of the classes are appended to form the name of the association, in this case *Coins_Control*. Moreover, none of the associations has their roles defined. The processor therefore assigns the default roles are “G1”, “G2”, etc. Finally, all spaces that occur in names are removed, so *Ware Mgr* becomes *WareMgr*.

IV 3.2 Hierarchical Structure

The classes and aggregations of an UML class diagram are translated into a static SDL structure specification. If the “generate types” option is on, a class is translated as a block type or process type. An aggregation is then used to identify an instance of the block or process type. If a class with stereotype «block» owns a state diagram, a process is created in the corresponding block. Figure IV-8 shows the resulting hierarchical structure after translating the class diagram of the toffee vendor (Figure IV-7). There are two block instances at system level, one for each top-level «block» class: a_WareMgr and a_Dialogue. These are the default names for instances, formed by prefixing "a_".

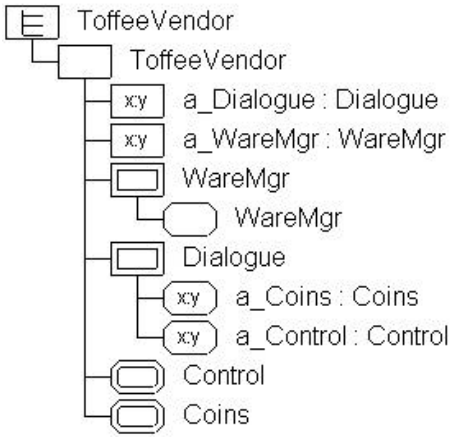

Figure IV-8. Hierarchy View of the Generated System

IV 3.3 Declarations and Communication

Signal declarations are an important aspect of an SDL system, as they form the only way of communication. Each signal used within a process, either as input or as output signal, should have a declaration within the scope of that process. An option of the translation lets the developer choose whether to put the signal declaration at system level or at the deepest block that is visible to all processes that use the signal. The same holds for type declarations. In our example, we choose to put all signals at system level.

Figure IV-9 shows the signal and type declarations in the toffee vendor system. Note that, besides the signal declarations, a signal list is created for each class that contains all the signals the class can receive. These signal lists makes it much easier to maintain the SDL specification manually, because adding a signal only requires one update instead of many throughout the system. Two empty newtype declarations are created for the «newtype» classes *Item* and *Cont*.

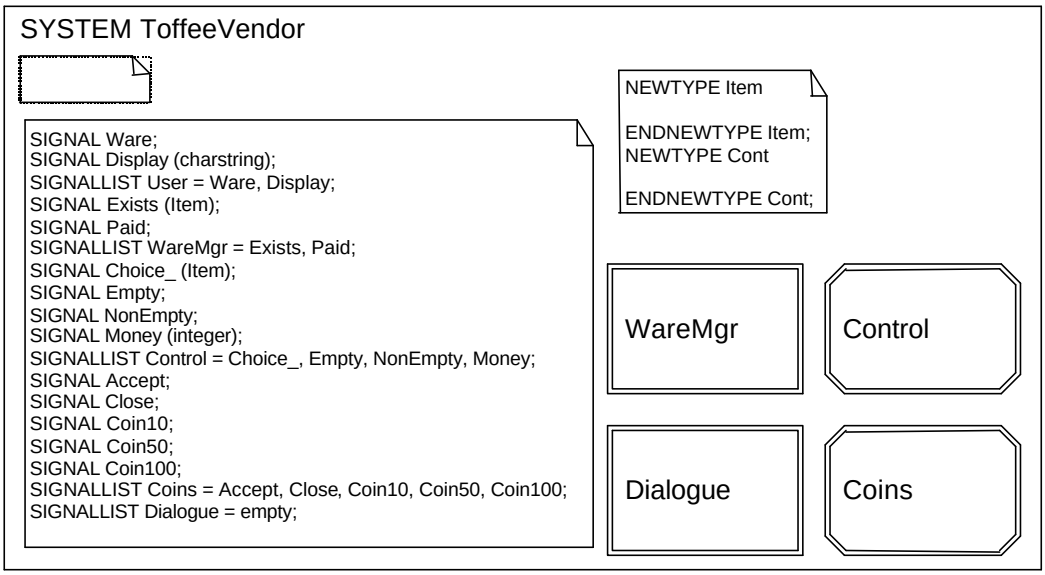

Figure IV-9. Signal and Type Declarations in the Generated System

Another important issue is to connect processes that communicate through channels and signal routes. For every association between two classes, a communication path is generated between the corresponding processes or process instances. Channels are generated to reroute the communication path via the first common visible block. By default, the channels are made bi-directional and hold the signal lists of the source process and destination process.

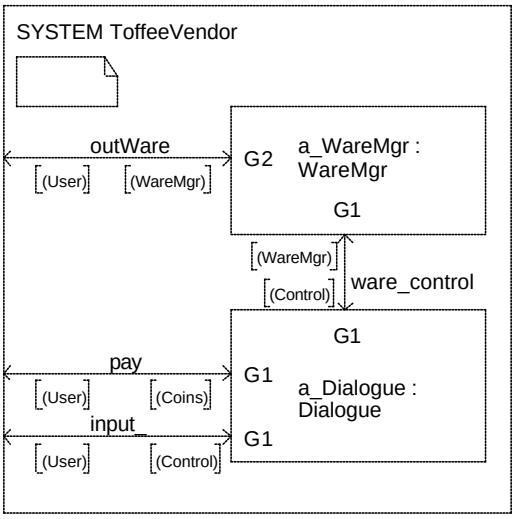

Figure IV-10. Block and Processing Interaction

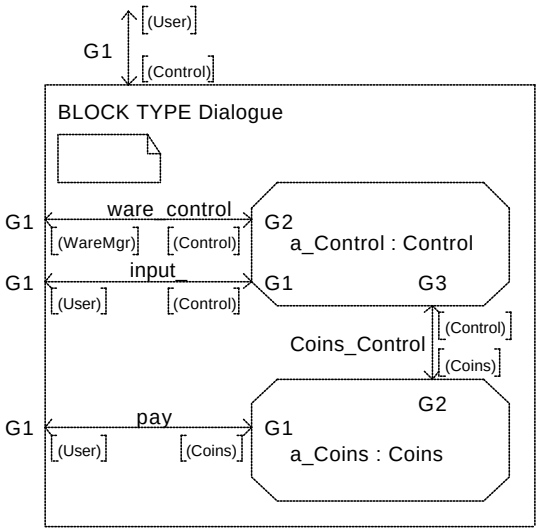

Figure IV-11. Process Interaction in Dialogue Block Type

Figure IV-10 shows the block interaction at the system level after having translated the complete UML model. The channel *ware_control* is the result of rerouting the association between the classes *Control* and *Ware_Mgr*. The three other channels going to the environment are the result of the associations to the *User* class.

IV 3.4 Finite State Machine

Each UML state diagram is translated into a finite state machine (FSM). Both «process» and «block» classes can contain a state diagram, making it an active class. The FSM is created in the process that is generated from corresponding class. In the case of a «block» class, an additional process is created in the block (type) that holds the FSM and the variable declarations.

Most constructs in an UML state diagram can directly be mapped onto SDL. States and transitions are equivalent in UML and SDL. Input events become input signals, output events become output signals and guards become provided constructs. Concerning actions, we differentiate four kinds of actions: assignment, output event, function call and informal text. Each of them is automatically recognized and accordingly translated to SDL. The other UML state diagram features are first convert to the basic features. Nested state diagrams are flattened before the translation. At the same time, entry and exit actions are moved onto the transitions. Figure IV-12 shows the finite state machine after translating the state diagram of the *Control* class shown in Figure IV-3. Note that the entry actions of the *payment* state are put on the *NonEmpty* transition and the exit action of the same state is put on the last transition.

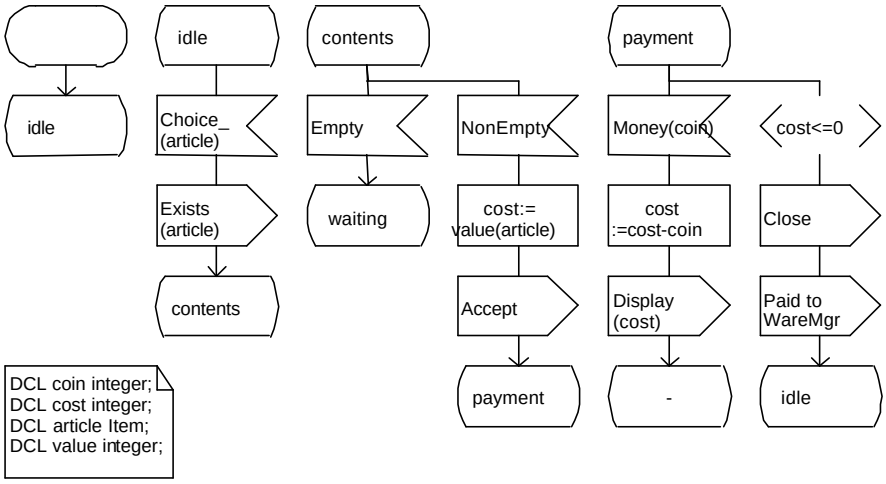

Figure IV-12. Generated FSM for *Control_Process*

IV 3.5 Linking UML and SDL models

In order to perform incremental changes after the initial code generation, we maintain links between the UML model and the SDL specification. All UML entities contain a number of link-variables that are specifically used to link the SDL entities that are generated from the entity. The set of link-variables depend on the UML entity. For example, a class is linked with the generated signal list, block and process. An association is linked with all generated channels, signal routes and gates. An operation is linked with the signal declaration, and so forth. The complete definition of this process is presented in section V. 5. Figure IV-13 illustrates the links built up during the translation of the toffee vendor. It shows a representative selection of entities of the UML model and SDL system in a hierarchical structure and the links between the corresponding constructs. The *Ware Mgr* class, for example, has three links: the signal list, the block type and the process.

The UML-SDL links are used during the incremental translation to locate the entities that are affected by the change. Suppose the *Ware Mgr* class is deleted, then the linked signal list, block and process are removed. If an attribute is added to the class, then a variable is added to the linked process. If the name of the class changes, then the linked entities are renamed too, etc.

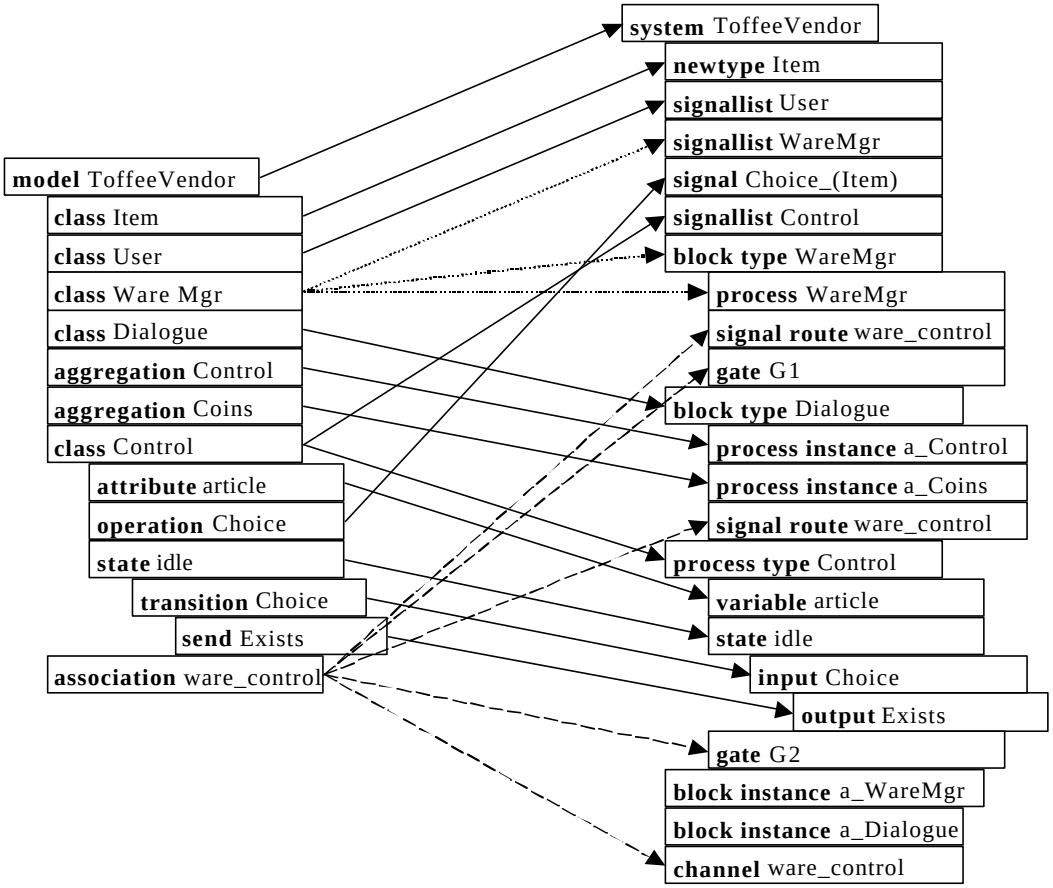

Figure IV-13. Hierarchical Links between UML and SDL

IV. 4 Detailed Design

If, after inspecting the generated SDL, the developer cannot or does not want to improve the UML at this point, he/she starts detailed design. In other words, he/she starts modifying and improving the generated SDL specification. On one hand, we lose the abstraction mechanisms of UML, but on the other hand, we gain the features of SDL that allows one to exactly specify the system. In this section, we discuss what typical detailed design embraces on the generated code and how it might affect the iteration. As a general guideline, structural changes are better performed in UML and local changes are better performed in SDL. Of course, a number of SDL constructs do not have a counterpart in UML and therefore always need to be design in SDL.

IV 4.1 Structures

Typically, the very first task in SDL is to improve the layout of the block and process interactions. For example, blocks are moved to avoid crossing channels or new declaration fields are created to group entities that belong to each other. Actually, the figures shown in the previous section were already modified in order to take less space. At the same time, superfluous declarations are deleted and the rest is reorder and grouped.

At this point, the block and process structure is fine-tuned. Many SDL specific issues cannot be expressed in UML and thus are performed during detailed design. In our example, we carry out three structural changes: we convert the *Dialogue* block type into a block, we let the *Control* process create a *Coin* process dynamically and we move the process types into the *Dialogue* block. Each of these changes is explained in more detail below. Figure IV-14 shows the resulting structure hierarchy.

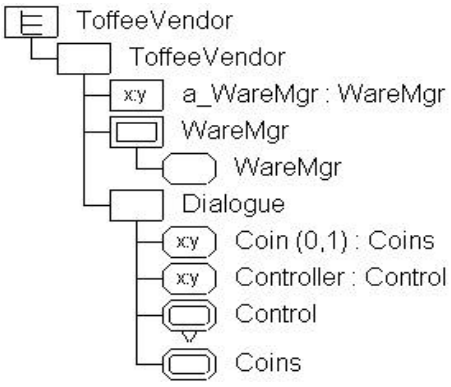

Figure IV-14. System structure after Detailed Design

The choice whether to translate classes as types or as definition is a global option and therefore needs some fine-tuning during detailed design. In our example, the only purpose of the *Dialogue* block type is to group the processes *Control* and *Coins*. We therefore transform the block type *Dialogue* into a block definition and delete the block instance *a_Dialogue*.

The default name for a type based block or process is the name of the type with "a_" as prefix. We give the process instances *a_Control* and *a_Coin* clearer names: *Controller* and *Coin.*

Furthermore, we modify *Coin* such that one instance is created dynamically, in our case by *Controller*. As a side effect, the *Control* process type needs the *Coin* process instance within his scope level. We therefore move the *Control* and *Coins* process types in the *Dialogue* block. The resulting structure is shown in Figure IV-14.

IV 4.2 Communication and Declarations

Wherever blocks or processes are moved or created, the channels and signal routes have to be adapted accordingly. As a guideline, its better for the round-trip engineering process to reconnect existing channels if possible. In this way, the link between the association and the channel is retained. In our example, the type based *Dialogue* block is replaced by the *Dialogue* block definition and all channels previously connected to the block instance are reconnected to the block definition. Moving the *Control* and *Coins* process type to a different place does not have an impact on the communication routes.

In the *control* process, we want more control over the destination of the signals it sends. To access the *coin* process, which is created dynamically, we create a variable *coins* that stores the PId after creating the process. To access the ware manager, we rename the rename the gate *G2* of the Control process type into the more meaningful name *Ware* such that we can use it in a via statement. The *ware_control* signal route is adapted to connect with the renamed gate. One more change is to remove the signal list of the *Dialogue* block, which is never used.

A detailed design activity is filling in the newtype declarations. The translator only handles structural newtypes, while in our example, the *Item* newtype is an enumeration of literals and the *Cont* newtype is an array. The specification of these two newtype is shown in Figure IV-15.

```
NEWTYPE Item
  literals toffee, chocolate, gum;
  operators value: Item -> integer;
ENDNEWTYPE Item;

NEWTYPE Cont
  array(Item, integer)
ENDNEWTYPE Cont;
```

Figure IV-15. Detailed design of newtypes

IV 4.3 Dynamic Behavior

Adding detail to the process specification is probably the most important aspect of detailed design. Here, issues like timers, addressing and switches are tackled. On the initial transition, processes can be created and their addresses queried and stored. Output events can then be modified to send signals to specific processes. Tasks can be added or improved. Transitions can be grouped together using the decision constructs. In addition, error handling can easily be modeled in SDL using the asterisk state and priority input constructs.

Figure IV-16 shows the Control process of our example after detailed design. The gray areas indicate the changes compared to Figure IV-12. On the initial state transition a process *Coins* is created, the output events have changed and the two transitions of the payment state have merged. The state *idle* is renamed into *waiting*. The destination of the output signals is changed from the class name into the correct PId value or gate. Furthermore, a new procedure *IntToString*

is added that converts an integer into a string. This procedure is called to convert the actual parameter when outputting *Display*.

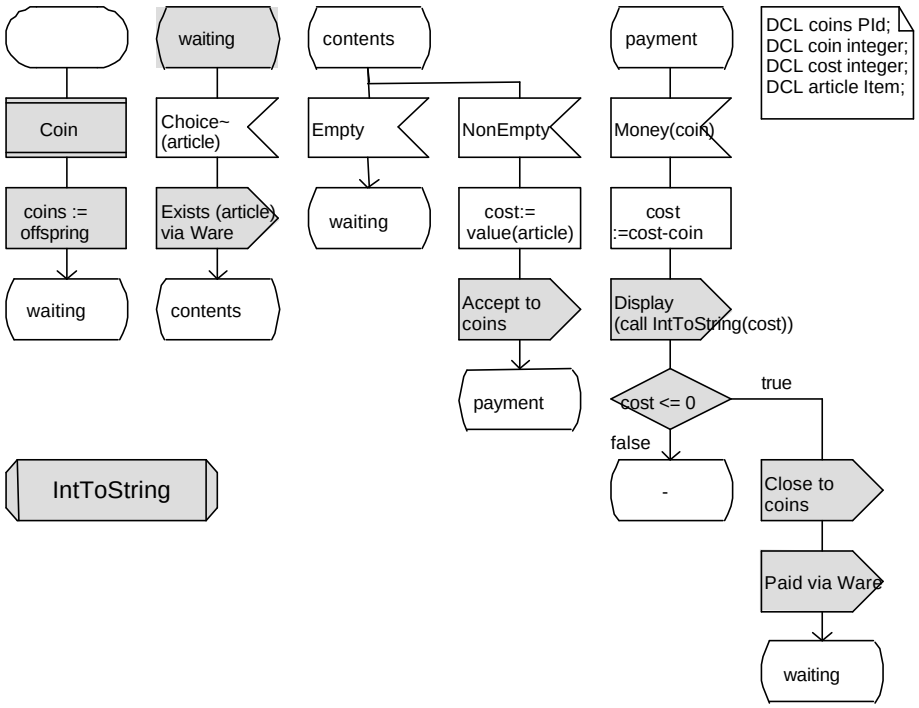

Figure IV-16. *Control* Process after Detailed Design

IV 4.4 Reverse Iteration

Before switching to system design in UML, the relevant detailed design changes are translated to the UML model. As the major part of detailed design deals with SDL specific things, many of the detailed design changes will not be translated at all. The changes are detected by comparing the previous version of the SDL system (in this case the generated SDL) with the last version. Entities that appear in the old version and not in the new version are deleted. Entities that do not appear in the old version, but do appear in the new version are added. Entities that appear in both versions are compared to search for differences in name, type, scope, inheritance, etc. The comparison can also be hand tuned, e.g. to join a *delete* and a *new* operation into a *modify* operation. Some more details on the comparison and incremental translation is given in the forward iteration step below. The full definition is specified in chapter V.

Table IV-1 shows the list of changes found in the SDL specification and the corresponding translation in UML. Note that in the first change, the structure link of the *Dialogue* class that original pointed to a block type is updated to point to the *Dialogue* block definition.

SDL Change	**Translated change in UML**
Block Type *Dialogue* transformed to Block	Class *Dialogue* becomes non-typed, update the

	structure link (ref. IV 3.5)
Block Instance *Dialogue* transformed to Block	No change (the aggregate defines the scope instead of the instance)
Process instance *Coin*, number of instances changed	No change
Process types *Coins* and *Control* moved	No change
Process instances *a_Coins* and *a_Control* renamed to *Coin* and *Controller*	Rename role names of aggregations
Gate *G2* of Control process type renamed to *Ware*	Rename role of association *ware_control*
Channels *ware_control*, *pay* and *input~* reconnected to *Dialogue* block	No change (associations connected to the same class)
Procedure *IntToString* added	Add operation to *Control* class
Implement newtypes	No change
Rename state *idle* into *waiting*	Rename the state
Action *create Coin* added to initial transition	Add informal action to initial transition
Action *coins := offspring* added to initial transition	Add assignment action to initial transition
Signal outputs modified	No change
Transitions from state *payment* merged (delete guarded transition.	Delete transition from payment to idle, but we ask not to apply the change.

Table IV-1: Detailed design changes in SDL and translation in UML

Updating the UML model concludes the first iteration. The first iteration is a special case because the incremental UML to SDL translation actually generates a complete SDL specification. The UML model is virtually compared with an empty model such that all entities are considered new. In the second step, i.e. detailed design in SDL, we mainly focus on the SDL specific issues. Consequently, the reverse incremental translation does not change a lot at the UML side.

IV. 5 System Design II

In this second iteration, we add new functionality to the toffee vendor. More specifically, we will improve the interaction with the user. We switch back to the updated system design model without loosing the detailed design. We discuss the changes in the class diagram and the controller's state diagram. In the diagrams below, the changes resulting from reverse iteration of the detailed design changes are marked with the tiled pattern.

IV 5.1 Class Diagram

As a continuation of our system design of the toffee vendor, we now include a class *Viewpoint* to the class diagram. *Viewpoint* is a class that handles the interaction with the user. It senses when a button is pressed and it shows messages on the display, e.g., how much money needs to be entered. The *User* actor now communicates with *Viewpoint* instead of the *Control* class. One end of the association *Input* is moved from *Control* to *Viewpoint*. Another association *displ* is added between *Viewpoint* and *Control*. Finally, an operation *Complete* is added as a notification from the *Ware Mgr* that it has delivered the item. The resulting class diagram is shown in Figure IV-17. The system design changes compared to Figure IV-7 are marked in gray; the detailed design changes are marked with the hatched pattern.

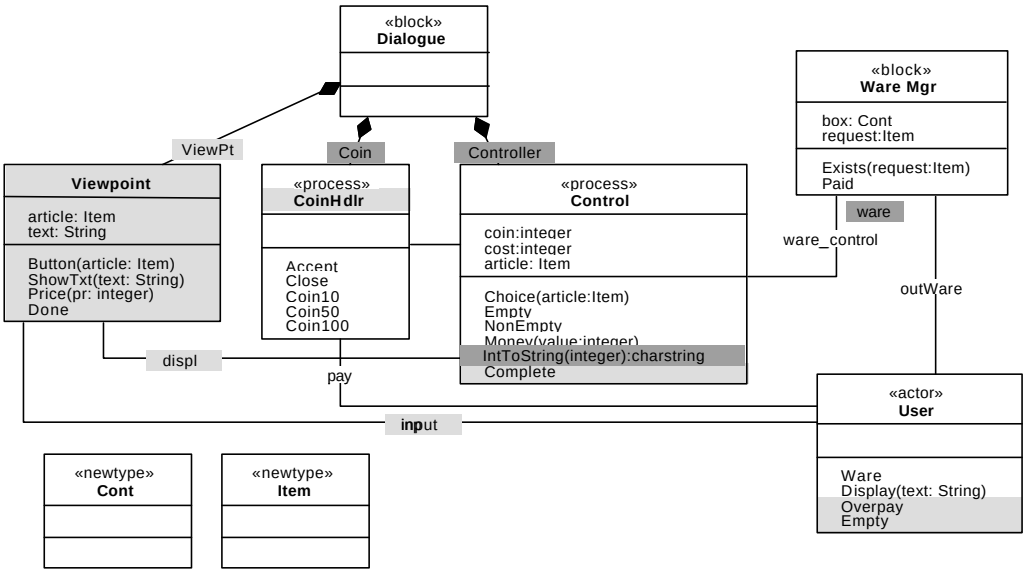

Figure IV-17. New System Design Model

IV 5.2 State Diagram

For the dynamic model, we only discuss the changes in the state diagram of the *Control* class. Most changes are related to the communication with *Viewpoint*. The signals *Showtxt*, *Price* and *Done* are all sent to *Viewpoint*. In the internal transition *Money(coin)*, the output of *Display* is replaced with the *Price* signal. An extra state *releasing* is needed to wait for the event *Complete*

from the Ware Mgr. Figure IV-18 shows the resulting state diagram. Of course, the state diagram of the classes *Viewpoint* and *WareMgr* are changed too, but are not further discussed.

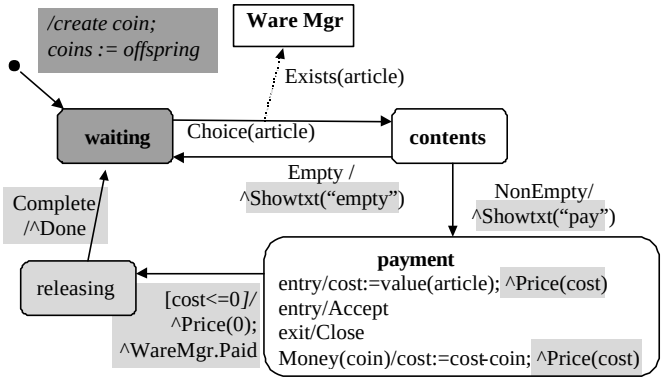

Figure IV-18. New State Diagram of Control

IV 5.3 Forward Iteration

After having changed the system design model, the detailed design model needs to be synchronized. Similar to the reverse iteration, this is accomplish by translating the changes in the system design model and apply them to the detailed design model. Unlike the reverse iteration, the UML changes in this forward iteration step have many implications on the SDL specification. We therefore discuss this step in more detail. The problem is threefold. How do we detect the system design changes (deltas), how do we translate them and how do we apply them to a specification that may be different than we expected.

IV.5.3.1 Determine delta's

The first aspect is to determine the deltas between two UML models, i.e. determine the changes made to the model during system design since the previous iteration. For this purpose, we compare the models before (old model) and after editing (new model). Whenever (re)entering a certain phase, the current model is stored before editing. After having changed the model, it is saved as a second version and then a standalone program compares those models. If an entity is present in the old model and not in the new model, the entity has been deleted. If an entity is only present in the new model, the entity has been added and is considered "new". If an entity is present in both the old and the new model, the entities "match" and are further compared to detect differences on a lower level. The matching of entities is based on their unique identifiers, such that renamings can also be detected.

The multi-level comparison follows the hierarchical structural of UML. At top-level, the incremental translator checks whether classes or associations are added or deleted. For matching classes, the attributes, operations, generalization, aggregates and state diagram are compared. Within the state diagram, the states are compared and on the transitions on the next level. As for our example, the changes found during the comparison are discussed together with the translation in the next section.

An alternative approach to determine changes is to track every edit commando during editing, together with its context. It is preferable to have a structural editor in order to store useful context information during tracking. Most graphical editors, however, are already structural editors.

Although we have chosen the comparison approach, the rest of the iteration is independent of this choice, as both approaches produce a set of changes.

IV.5.3.2 Translate and Apply Deltas

The essence of the iteration is of course the actual synchronization of the system design model and the detailed design model. The changes detected during system design are translated and applied to the SDL specification. We go through the changes discussed in sections IV 5.1 and IV 5.2 and translate them in the meantime. A complete specification of the incremental translation rules is given in V. 6.

Viewpoint is a new class without a stereotype definition and is a component of the *Dialogue* class. The default stereotype for a leaf class is «process». This change is translated by creating a new SDL process type named *Viewpoint* and adding it to the *Dialogue* block. The instance of the process type is created later with the translation the new aggregation. Figure IV-19 shows the content of the Dialogue block after applying the incremental translation.

The «process» class *Coins* is renamed into *CoinHdlr*. This is translated by renaming the linked process type *Coins* into *CoinHdlr*. The process instance is not renamed, as the explicit role name of the aggregation is not changed.

The attributes and the operations in the *Viewpoint* class are all new too. Compliant with the mapping, the new attributes are translated as new variables in the *Viewpoint* process type and the new operations are translated as new *signals*, which are added to the signal list. The new operations in the classes *User* and *Control* are translated the same way.

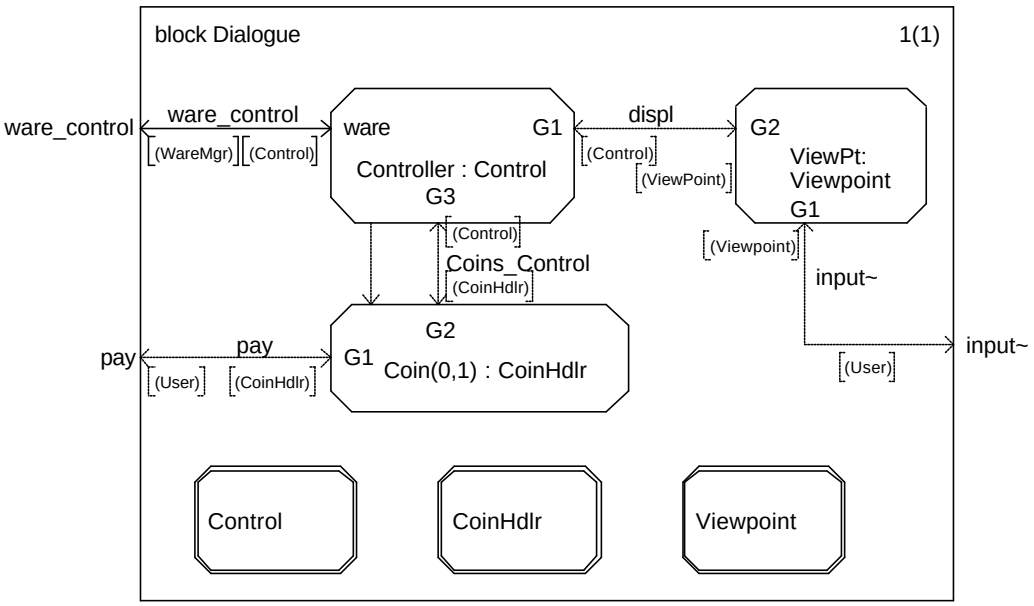

Figure IV-19. Static Structure of *Dialogue* block after Forward Iteration

A new association called *displ* is added between *Viewpoint* and *Control*. This is translated by adding a gate the *Viewpoint* and *Control* process types and by connecting the process instances with a signal route. Note that the order of executing the changes in important, as this translation needs the generated *Viewpoint* process type.

The association *Input* is connected to a different class; more specifically, the *Control* association end is moved to *Viewpoint*. In theory, it would be sufficient to regenerate only one side of the communication routes. However, it is very difficult to specify exactly which part of the channels and signal routes of the unchanged side are still valid (gates, used signal lists, connections, etc.) Therefore, the full communication route is regenerated in cases like this. To demonstrate the incremental translation of state diagrams, we discuss the changes in *Control's* state diagram. To find the changes, we compare the old version of the state diagrams, shown in Figure IV-3, with the new version, shown in Figure IV-18. The old version also includes the two actions on the start transition, which are generated during reverse iteration. These actions are therefore not considered new.

In five transitions, a new action was added; all of these actions are output signals. These changes are translated by adding an output action at the end of the corresponding transitions. There is one exception; during detailed design, the guarded transition starting from the payment state was merged with the *Money* transition. Consequently, the UML transition between the states *payment* and *idle* is not linked anymore and the translations cannot be applied.

The result is shown in Figure IV-21. The hatched areas are changes already applied during the detailed design phase, while the gray areas mark the changes applied by translating the system design changes.

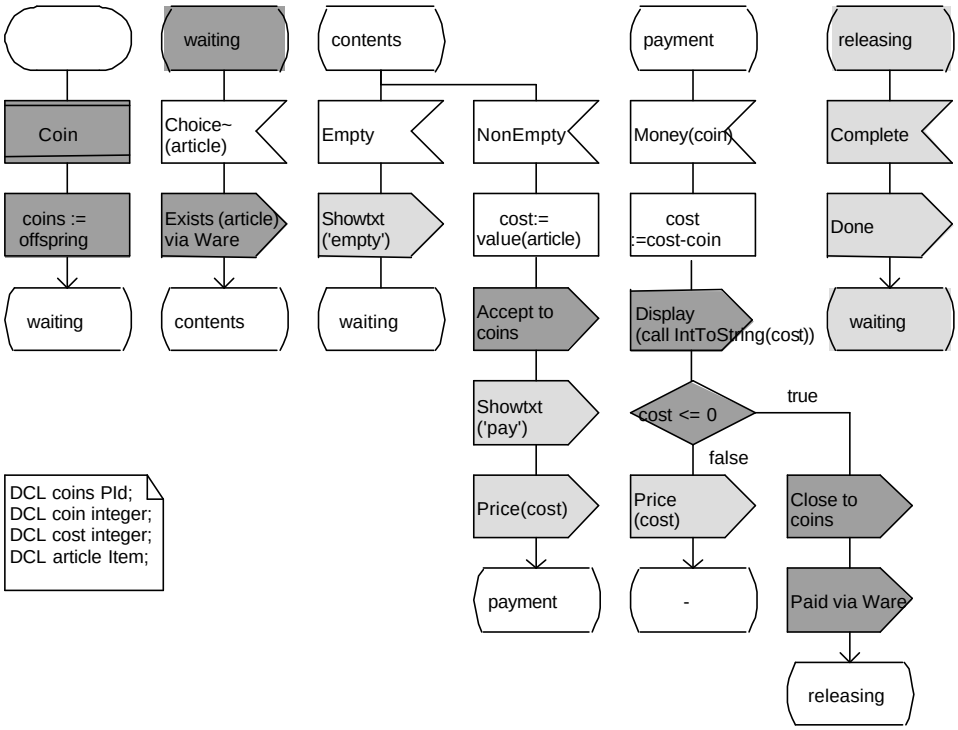

Figure IV-20. *Control* Process after forward iteration

IV. 6 Detailed Design II

The second iteration continues with the adaptation of the updated SDL specification. We manually finalize the translation of system design changes and we implement new functionality that is difficult to achieve in UML.

On the structural and communication level, the translated system design changes do not need further detailed design. For the state machine, we again focus on the *Control* process. First, we apply the changes that could not be added because of the missing link. More specifically, we add an action to send *Price(0)* to the *Viewpoint* process and we correct the destination state of the transition. Second, we add some new functionality. To avoid deadlocks, we build in a timer that gives the user of the toffee vendor a certain amount of time. If the time is exceeded, the sale is canceled. The user is also given the possibility to *undo* the sale himself. The resulting process definition is show in Figure IV-21, only the relevant states and transitions are shown. The new detailed design changes are marked in gray. The hatched areas represent all the previous edit operations (system design and detailed design).

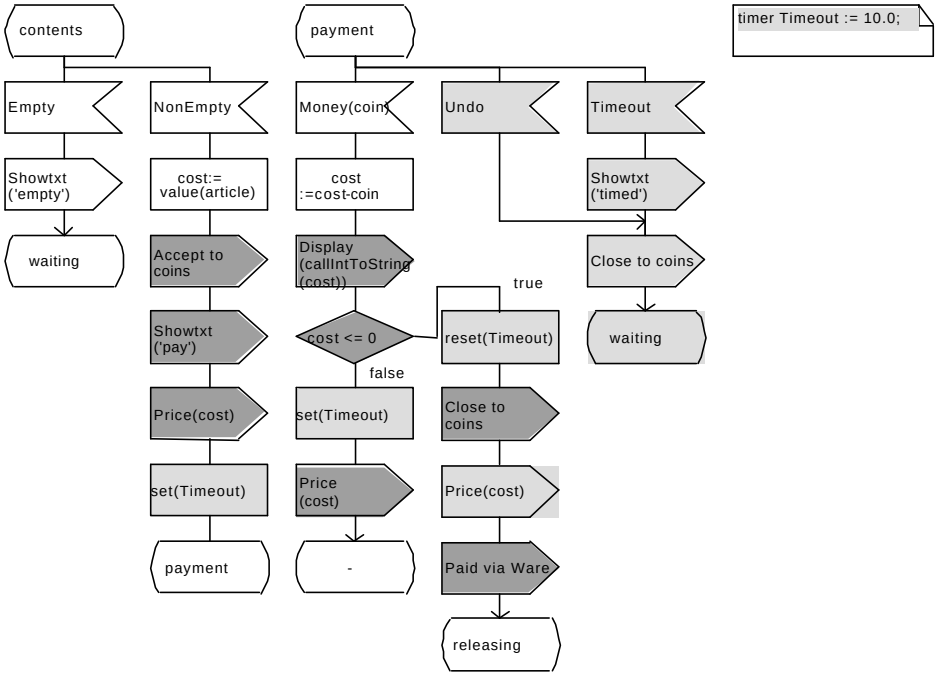

Figure IV-21. *Control* Process after Forward Iteration

Again, these changes are translated into incremental changes to the UML model. There are two new transitions from the payment state to the waiting state. The actions below the join of the *Undo* transition are duplicated, such that the actions are present in both transitions. The new transitions are translated by adding two transitions to the UML state diagram. The resulting state diagram is shown in Figure IV-22 with the new elements marked in gray.

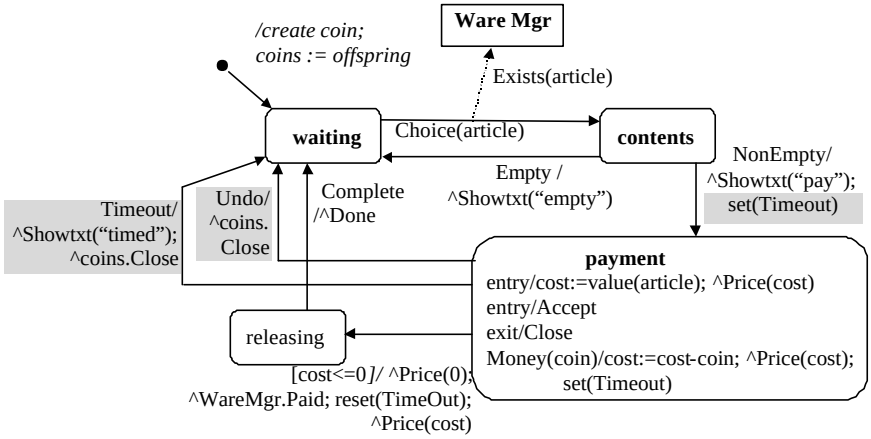

Figure IV-22. State Diagram of Control after Second Iteration

This step concludes our example. This loop of system design and detailed can be repeated any number of times. After each iteration, the updated model or specification is examined and improved. After a few iterations, the UML model and SDL specification may diverge somehow. This has the advantage that the UML model can present a higher abstraction level. The disavantages are that UML and SDL are not completely synchronized and that due to missing links there is less support for iterative translation. The latter can be tackled by a manual relinking process.

V. REALIZING THE UML-SDL ROUND-TRIP ENGINEERING

“Act as if what you do makes a difference. It does.”

-William James-

“Good timber does not grow with ease. The stronger the wind, the stronger the trees.”

-Williard Marriott-

V. 1 Introduction

This chapter contains the complete definition of the UML-SDL round-trip engineering process and is the technical core of this dissertation. A large set of rules defines how a UML model is preprocessed and how all the possible changes in UML or SDL are translated into changes in the opposite model. Before executing any rules, we need to build a valid information model of the UML model and the SDL specification. Moreover, the two information models must be linked with each other to determine the right scope for translating changes. The next subsection gives an overview in which order the various activities are executed, how the data flows between the activities and which section in this chapter deals with these issues. Section V 1.2 explains the notation of the transformation rules used throughout the chapter.

V 1.1 Overview of the Round-Trip Process

Figure V-1 captures the flow of activities and models during the round-trip process as described in this chapter. The process starts by loading the old and the new versions of the UML model. In the first iteration, the old UML model is empty and all entities in the new model are considered new. In the succeeding iterations, the old model is the previous version of the UML model. The exact structure of the UML information model is defined in section V 2.1. Both the old and the new UML model are preprocessed to check their consistency and to fill in missing information with defaults. The preprocessing rules are defined in section V 2.2. On the SDL side, the latest version of the specification is loaded in a specialized information model defined in section V. 2. Next, the links between the old UML model and the SDL specification are restored. Essentially, each UML entity has pointers to the SDL entities is has generated. Section V. 4 defines the exact list of links and back-links. The links between the new UML model and the SDL specification is build-up and updated during the compare process.

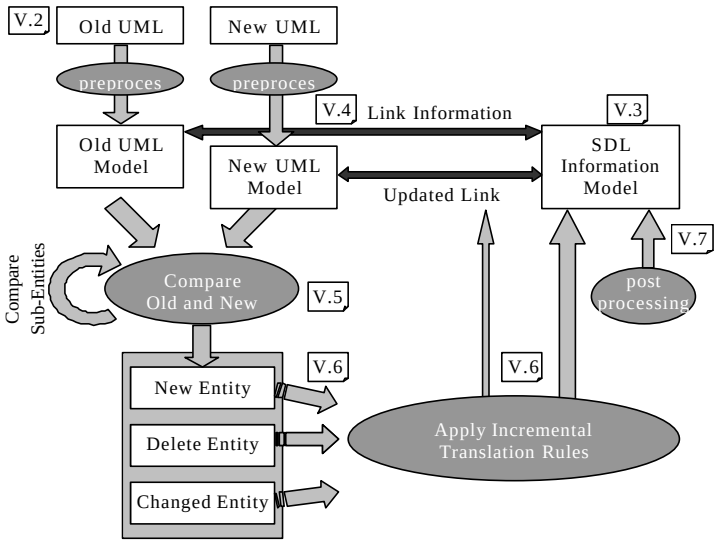

Figure V-1. Overview of the Forward Incremental Process

Once all data structures are in place, the old and the new UML models are compared to detect changes made in the model since the previous iteration. If an entity is present in the old model, but not in the new model, the entity has been deleted. If an entity is not present in the old model, but is present in the new model, the entity has been added (*new entity*). If an entity is present in both models, they are said to "match". Matching entities are compared on their attributes, e.g. name, stereotype, etc. The comparison is done in a hierarchical fashion. First, the packages are compared, then the components and associations and then the attributes, operations and state diagrams. All sub-entities of a new entity are also new and all sub-entities of a deleted entity are also deleted. The order in which the changes are handled is crucial, because some changes interfere with each other. Section V. 5 defines the exact order in which the comparison is executed.

For each new, deleted or matched entity, a set of rules is applied to the entity. Except stated otherwise, the rules are applied in order of appearance. Each rule defines a precondition that must be fulfilled before the action part is actually executed. The rules are defined in section V. 6, which is divided in subsections. Each subsection contains the set of rules necessary to translate or compare a particular entity.

The second part in the round-trip engineering is the reverse incremental translation of changes in SDL to update the UML model. The process, illustrated in Figure V-2, is almost identical to the forward incremental translation. In this case, there are two SDL specifications that are compared and one UML model that is being updated. The link between UML and SDL is restored based on the old SDL specification. However, because the links are stored on the UML side and are based on identifications instead of direct pointers, there is also a link between UML and the new SDL specification. As in forward iteration, the links are updated to the new SDL specification in the translation rules. The rest of the process is identical to the forward variant, even though we use different comparison and translation rules (see section V. 8).

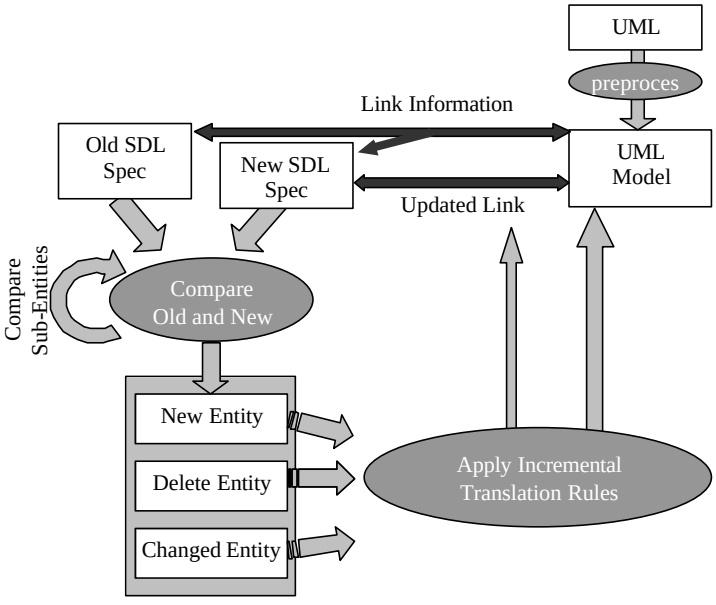

Figure V-2. Overview of the Reverse Incremental Process

V 1.2 How to read rule definitions

The UML preprocessing and the translation of changes is defined in the form of rules. Each rule describes the actions to be taken if a certain condition is satisfied. The rules are presented in a uniform table with a maximum of four fields, as illustrated in Rule 0. The precondition field provides a list of conditions. If not mentioned otherwise, *all* the conditions must be satisfied, for the rule to be executed. If there is no precondition field, the rule is always applied. The context field provides a list of variables within the context of the rule. They can be regarded as a number of "let" statements. The action field defines the actions that are taken when the rule fires. Most rules for translating new UML entities add a number of SDL constructs to the SDL data structure. The SDL is presented in the textual representation for easy readability. For example, instead of writing "`system.addStructure (new SdlBlockType(`*var*$_2$`));`", we write the statement shown in the illustration (Rule 0). The variable field, finally, defines a number of variables that are used by other rules in the same section or fills-in UML-SDL links.

A general principle in all rules is that every variable, function or link that contains or returns an SDL value is underlined. Variables, functions or links that contain or return an UML value are not underlined. This notation increases readability because most rules mix UML and SDL expressions.

Preconditions	– *Condition 1* – *Condition 2*	All conditions must be fulfilled to make this rule fire.
Context	– var_1 = *UMLVariable* – var_2 = var_1.*umlattribute* – $\underline{var_3}$ = var_1.*sdllink* – $\underline{var_4}$ = *SDLVariable* – var_5 = $\underline{var_4}$.*reverselink*$^{-1}$	Everything that contains or returns an SDL value is underlined.

Action	Do something.
	Add to system $\underline{var_4}$:
	`BLOCK TYPE <`var_2`>;`
	`ENDBLOCK TYPE <`var_2`>;`
		References to variables.
		Generated code.

Variables	– *Global Variable* = new value	Definition of new variable or assignment of new value

Rule 0. Illustration of a rule.

V. 2 UML Information Model

The first activity in an iteration step is loading the UML model in a specialized information model and preparing the UML model for its translation to SDL. This section defines exactly what information is used for the translation and how it is organized. Almost any UML tool allows more information than can be stored in our information model. This extra information is simply ignored, as it is of no importance for the UML-SDL round-trip engineering. On the other hand, some information specific for the round-trip engineering is added to the model. This information is filled in during the preprocessing or during the translation. The next section discusses the UML information model in more detail.

V 2.1 Information Model

Figure V-3 shows an overview of the entities in the UML information model. The tables below define the attributes for the different UML entities. The first column gives the name of the attribute. The second column describes the semantics of the attribute or defines the possible valid values. The description also indirectly defines the type of the attribute. The last column defines the default value in the case that the attribute is empty. Filling in the default is one aspect of the UML preprocessing. We discuss some important issues for each of the entities.

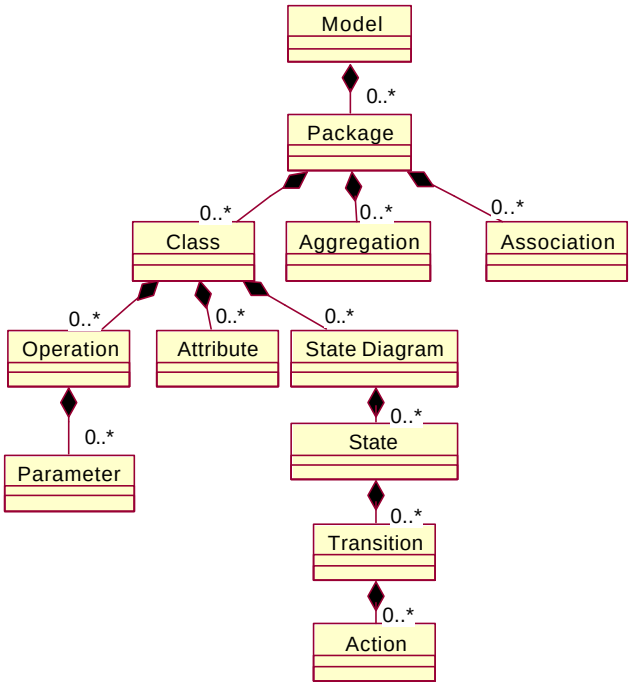

Figure V-3. UML information model

The *model* is the root node for a UML model. Except for the name, it contains a flattened list of all *packages* in the model, i.e. the list also contains the packages that are nested within other packages.

UML Model	Description	Default
name	Usually corresponds with file/project name.	"the_model"
packages	List of packages	-

Table V-1. UML model

The *package* is the main structuring mechanism. The stereotype determines whether the package maps on an SDL system or on an SDL package. The *name* does not have a default, a package without name is invalid and is skipped. The *system class* attribute points to a special class that represents the package. The system class is created during preprocessing and is an aggregate of all top-level classes by construction. The attributes *global declaration* and *global type* are both translation options. They affect in which scope declarations and structured types are defined. The *packages* attribute points to the packages in the model that this package depends on.

UML Package	Description	Default
stereotype	«system» or «package»	«package»
name	An empty name is invalid	-
system class	The class that represents the package or system	created during preprocessing
global declaration	if true, all signal and newtype declarations are defined at top level, if false: local declaration	true
global type	if true, all block/process types are defined at system/package level, if false: local declaration	false
communication	Option for generating communication, possible values are: *none*, *gate only*, *conservative*, *full*	conservative
packages	The list of package that this package depends on	-
classes	The list of all classes in the package	-
associations	The list of all associations in the package	-
aggregations	The list of all aggregations in the package	-

Table V-2. UML package

The UML class is the most complex entity as it is overloaded with different semantics, depending on the *stereotype*, which is the first attribute of the UML class. The stereotypes «system» and «package» are reserved for the system class of the package. A class with stereotype «block» is mapped on an SDL block or block type with an optional process containing the behavior of the class. A class with stereotype «process» defines an active entity with its own variables and state diagram and is mapped on an SDL process or process type. A class with stereotype «newtype» describes an abstract data type, which can be used to type attributes and parameters. A class with stereotype «actor», finally, serves as placeholder for an active entity outside the system. Not all the attributes are applicable for all kind of classes.

The *class name* is essential for all classes and must never be empty. The attribute *is extern*, is true for classes that are imported from another package. «Actor» classes are always external. For external classes, the *package name* attribute defines in which package the type is defined. A «block» or «process» class can contain maximum one *state diagram*. The *typed* attribute indicates whether a «block» or «process» class is a structure definition or a structure type that will

be instantiated with aggregates. The attribute *defined in* points to the class in the same package that provide the scope for the type declaration, i.e. the structure type will be placed in the structure linked with the *defined in* class. The *management process* decides whether a «block» class is also mapped on a process or not.

UML Class	Description	Default
stereotype	Valid stereotypes: «system», «package», «block» «process», «newtype», «actor»	«block» or «process»
class name	An empty name is invalid	-
is extern	Is true for imported classes from another package and for actors.	false
package name	Name of the Package for external classes.	""
state diagram	The UML State Diagram of this class.	empty
typed	If true, class maps on SDL block or process type. If false, maps on block or process. Only applicable for «block» or «process» classes.	true
super class	Class where this class inherits from. If empty, no inheritance. Multiple inheritance is not supported.	empty
defined in	Class where this class is in defined.	system class
management process	If true class (also) maps on an SDL process. Always true for «process» class. True or false for «block» class. Always false for other classes.	true for «process», false for others
attributes	List of attributes of the class	empty
operations	List of operations of the class	empty

Table V-3. UML class

The UML operation has three different semantics depending on the stereotype. A «signal» operation declares a signal and at the same time defines that the class containing the operation can receive the signal. A «procedure» operation defines the signature of an SDL procedure. «procedure» operations can only be used in «block» or «process» classes. An «operator» operation, finally, defines the signature of the behavior for an SDL newtype.

UML Operation	Description	Default
stereotype	«signal», «procedure», or «operator»	«signal» («operator» for «newtype» classes)
name	An empty name is invalid	-
return type	Name of the return type	""
parameters	List of parameters of the operation	

Table V-4. UML operation

UML Parameter	Description	Default
stereotype	«process» or «block»	«package»
name	An empty name is invalid	-
type	Name of the type of the parameter.	"a_" + parameter name

Table V-5. UML Parameter

UML Attribute	Description	Default
stereotype	Not used in the translation	«»

name	An empty name is invalid.	-
type	Name of the type of the attribute.	"a_" + attribute name, see preprocessing options.
default	The string that represents the default value for the attribute.	""

Table V-6. UML Attribute

Our representation of aggregation and association relationships contains only the basic features: name, pointer to the connected classes, the role of the classes in the relationship and for aggregations a "composite" attribute. Other information like multiplicity, public/private and constraints are not needed for the translation to SDL. The default role name for associations is automatic counter prefixed with the letter "G".

UML Aggregation	**Description**	**Default**
name	May be empty.	""
aggregate	Pointer to the aggregate class	-
component	Pointer to the component class	-
aggregate role	Role of the aggregate class	""
component role	Role of the component class	"a_"+component name
composite	If true, composite aggregation. If false, reference aggregation.	true

Table V-7. UML Aggregation

UML Association	**Description**	**Default**
stereotype	«communication» or other	«communication»
name	Name of association	see preprocessing option
from class	Pointer to the "From" Class	-
to class	Pointer to the "To" Class	-
from role	String that describes the role of the from class	G#
to role	String that describes the role of the to class	G#

Table V-8. UML Association

The UML state diagrams consists of list of all states (including sub states) and a list of transition between those states. Sub-states have a reference to their super state. Start and exit states are regular states with the *type* attribute set to specific value. The UML transition contains many attributes to define its trigger: event, guard and timer. Table V-11 gives a short descriptions for each of the attributes.

UML State Diagram	**Description**	**Default**
name	Name of the diagram	Name of class
states	List of all states (including initial and termination states)	empty
transitions	List of transitions	empty

Table V-9. UML State Diagram

UML State	**Description**	**Default**
name	If this is a normal state, i.e. type=normal,	-

	then an empty name is invalid.	
super state	reference to the super state	empty
transitions	list of internal transitions	empty
type	Possible values: start, stop, normal	normal
entry actions	List of entry actions	-
exit actions	List of exit actions	-
activity	Name of activity	""

Table V-10. UML State

There are three types of transitions, which differentiate in when they are fired. An *event* transition is triggered by an incoming signal and can optionally be guarded with an expression. A *when* transition is fired when the guard expression becomes true, usually due to an assignment. An *after* transition is fired when a specific time has elapsed after entering a state.

UML Transition	**Description**	**Default**
event	Name of the signal that triggers the transition	""
source	Source State	-
dest	Destination State	-
type	Possible values: **event**, **when**, **after**	event
guard	Boolean expression that sets the guarded condition. Used for *event* and *when* transition.	""
timer	Time expression for an *after* transition	""
actions	List of UML actions	empty
is internal	Is true for internal transitions	false

Table V-11. UML Transition

UML Action	**Description**	**Default**
name	An empty name is invalid.	-

Table V-12. UML Action

V 2.2 Translation and Preprocessing Options

Several aspects of the UML to SDL translation are customizable. Here we define the options that are available to manipulate the translation and preprocessing. The options presented here are not accessed directly from the preprocessing rules and translation rules. Instead, the rules presume the default value of the options (with the exception of the *communication* option). The description below specifies what happens to the preprocessing or translation if the option is set to a different value. This way of working improves the readability of the translation rules, as the rules do not have to consider the different cases.

Option	**Description**	**Default**
Default Type	By default, the prefix "a_" is put in front of the variable name to fill in a missing type. If the *default type* option is different from "", than this type name is used to fill in the type of attributes or parameters without type.	""
Role Prefix	By default, associations without role definition get role name by prefixing "G" to a counter ("G1", "G2", ...). With this option, this prefix can changed.	"G"

Association Name	This option defines which name an association without a name is given. • *role*: compose role names <FromRole>_<ToRole> • *class*: compose class names <FromClass>_<ToClass>	*role*
Communication	This options define the way the association is translated into communication routes. This option has a great impact on the translation and is therefore considered in the translation rules. This option is referred to as the "*communication option*". The possible values are: • *no communication*: no translation, associations are ignored • *gate only*: max. two gates are generated • *conservative*: the association ends are translated, including the gates • *full*: a full connection is generated (possibly many gates & many channels/signal routes)	*conservative*
Avoid Management	If true, set management attribute of all «block» classes to false, so that no management processes (process linked with a «block» class) will be created.	*false*
Typed	This option influences the "typed" attribute of the classes in the model. If a class's typed attribute is true, it is translated as a block type or process type. Possible values of the *typed* option are: • *default*: no change, as defined in the class • *all true*: before preprocessing, make all «block» and «process» classes typed • *all false*: before preprocessing, make all «block» and «process» classes non-typed	*default*
Parameter Variable	If this options is set to true, a class attribute is created for each parameters of a «signal» operations. If false, Rule 16 is not executed.	*true*

Table V-13. Options for Preprocessing

V 2.3 Preprocessing

Before comparing, translating or synchronizing a UML model, the model is always preprocessed to prepare the model for translation. Doing the preprocessing as a separate step simplifies the translation rules, as they do not have to consider aggregation loops or empty names or types. It assures a consistent model with all missing information filled with defaults and it processes inheritance of association and hierarchical state diagrams. The name, type and/or stereotype of classes, operations, attributes and associations are filled with defaults if this information is missing in the model. Furthermore, it calculates the extra associations needed to make subclasses inherit the communication from their super classes, see V.2.3.5. The last major activity during preprocessing is the flattening of hierarchical state diagrams, see section V.2.3.6. Each of the sections below is executed once for each corresponding entity, e.g. Rule 2 is executed for every package and Rule 4 through Rule 10 are executed for every class in every package.

V.2.3.1 Preprocessing UML Package

The most important action for preprocessing a package is providing the *system class*, a special class that represents the package. The *system class* is used intensively during the whole translation process. It is constructed in Rule 2 by creating a new class with stereotype «package» or «system» and by making the class an aggregate of all active top-level classes. In other words, the system class becomes aggregate of all active classes that did not have an aggregate before.

Context	– *package* is the UML package to be preprocessed
Preconditions	– *package.stereotype* = ""
Action	*package.stereotype* = «system»

Rule 1. Default Stereotype for Package

Context	– *package* is the UML package to be preprocessed – *model* is the UML model containing *package*
Preconditions	– *package.systemclass* = empty
Action	*package.systemclass* = a new class with the following properties: *systemclass.stereotype* = *package.stereotype* *systemclass.name* = *package.name* ∀ *class* ∈ *package.classes* : if *class* has no aggregates and *class.stereotype* = «», «block» or «process», do Add an aggregation *aggr* to *package* with the following properties: – *aggr.aggregate* = *package.systemclass* – *aggr.component* = *class* – *aggr.composite* = true
Variables	*sysclass* = *package.systemclass*

Rule 2. Create system class

V.2.3.2 Preprocessing UML Class

The rules in this section checks and prepares the stereotype, inheritance relationship, typed property and the *defined in* property of each class. Note that a class can only have one super type because of the information type.

Context	– *class* is the class to be preprocessed.

Rule 3. Context for this section

Preconditions	– *class.stereotype* = "" – *class* has at least one components (is component part of a aggregation relationship) with stereotype «process»
Action	If class has a component *comp*, where *comp.stereotype* = «process», «block» or «», then *class.stereotype* = «block» else *class.stereotype* = «process»

Rule 4. Stereotype for every Class

Rule 5 and Rule 6 enforce that all classes involved in a inheritanc relationship are typed and have the same stereotype. Rule 7 checks whether a class has multiple aggregates, i.e. the class is the component part of an aggergation relationship. If this is true, it means that several instances are taken from the class and therefor the class should be typed.

Preconditions	– *class.stereotype* ¹ *class*.*superclass.stereotype*
Action	*class.superclass* = empty

Rule 5. Same stereotype for super- and sub-class

Context	– *class*.*superclass* is not empty
Preconditions	– *class.stereotype* = *class.superclass*.*stereotype* = «process» ∨ «block»
Action	*class.typed* = true *class.superclass.typed* = true

Rule 6. Classes with inheritance must be typed

Preconditions	– *class* has more than one aggregate – *class*.*stereotype* = «process» ∨ «block»
Action	*class.typed* = true

Rule 7. Multi-instance Classes must be Typed

Context	– *class* is a class.
Preconditions	– *class.typed* = false – *class* has exactly one aggregate
Action	*class.definedin* = aggregate of *class*

Rule 8. Non-typed Class are Defined in their Aggregate

Context	– *class* is a class.
Preconditions	– *class.typed* = true – *package.globaltype* = true
Action	*class.definedin* = *sysclass*

Rule 9. Global Typed Classes

Context	– *class* is a Class.
Preconditions	– *class.definedin* is not empty. – *class* has one or more aggregate – *package.globaltype* = false
Action	*class.definedin* = common aggregate of *class* (see description below)

Rule 10. "Defined In" for Local, Typed Classes

Rule 10 uses the term *common* aggregate to assign a default value for the *definedin* property of the class. The common aggregate of a class *comp* is the class *aggr* which scope contains all the instances of the class *comp*. Class *aggr* can be be found by comparing all the aggregation paths

(see section V 6.20) of *comp*. If only the first class (the system class) is common in all paths, then that is the common aggregate. Then the second classes in all paths are compared and so on. The last class which is common in all paths, except the class itself, is the common aggregate. In Figure V-4 the only aggregation path of class D is (A,B,D) so the common aggregate of D is B. Class F has two aggregation paths: (A,B,E,F) and (A,C,E,F). The second class in the two paths are different, so the common aggregate of F is A.

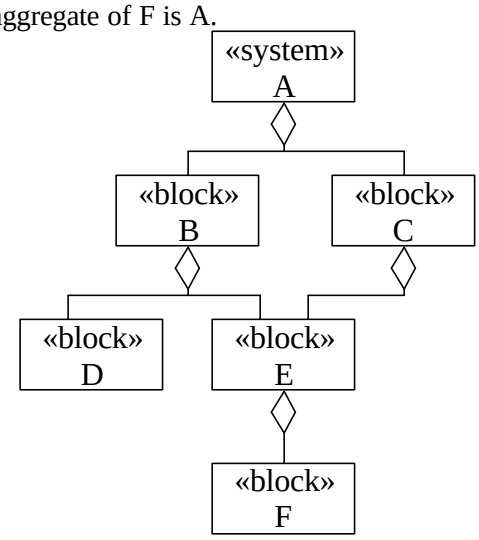

Figure V-4. Aggregate Structure to Find Common Aggregate

V.2.3.3 Preprocessing UML Operations

Context	– *operation* is the operation to be preprocessed – *class* is the class containing *operation*

Rule 11. Context for this section

Rule 12 determines the default stereotype for an operation. A UML operation can map on an SDL procedure or signal. Signals cannot have a return type, therefore the default stereptype for operations without return type is «signal». Procedures usually have a return type, therefore the default stereptype for operations with return type is «procedure».

Preconditions	– *operation.stereotype* = «» – *class.stereotype* ≠ «newtype»
Action	If *operation.returntype* is empty: *operation.stereotype* = «signal» else *operation.stereotype* = «procedure»

Rule 12. Stereotype for every Operation

Preconditions	– *operation.stereotype* = «» – *class.stereotype* = «newtype»

Action	operation.stereotype = «operator»

Rule 13. Stereotype for operation in «newtype» class

Preconditions	– *operation.name* = ""
Action	Delete *operation* from the model. Nameless operations are not allowed.

Rule 14. Delete attribute without name

Context	– Execute this rule for each *parameter* ∈ *operation.parameters*
Preconditions	– *operation.stereotype* = «signal» – *parameter.type* = empty
Action	– *attr.type* = "a_"+*parameter.name*

Rule 15. Create Parameter Variables

Preconditions	– *operation.stereotype* = «signal»
Action	∀ *parameter*∈ *operation.parameters*, add an attribute *attr* to *class* with the following properties: – *attr.name* = *parameter.name* – *attr.type* = *parameter.type*

Rule 16. Create Parameter Variables

V.2.3.4 Preprocessing UML Attributes

Rule 17 deletes attributes that does not have a name. However, most UML tools will already enforce to give a name to attributes, so this rule should not be harmful.

Context	– *attribute* is the attribute to be preprocessed
Preconditions	– *attribute.name* = ""
Action	Delete *attribute* from the model; nameless attributes are not allowed.

Rule 17. Delete attribute without name

Context	– *attribute* is the attribute to be preprocessed
Preconditions	– *attribute.type* = ""
Action	Set *attribute.type* = "a_"+*attribute.name*

Rule 18. Default attribute type

V.2.3.5 Preprocessing UML Association

Context for this subsection	– *association* is an Association.

Rule 19. Context for Preprocessing Associations

Preconditions	– *association.fromRole* = ""
Context	– i = gate counter of *association.fromClass*
Action	Set *association.fromRole* = G<i> Increment the gate counter of *association.fromclass*

Rule 20. Fill-in empty from-role name

Preconditions	– *association.toRole* = ""
Context	– i = gate counter of *association.toClass*
Action	Set *association.toRole* = G<i> Increment the gate counter of *association.toClass*

Rule 21. Fill-in empty to-role name

In SDL, inheritance is expressed between types, while communication is expressed between (type-based) instances. To prepare types for communication, they are provided with gates, but it is not possible to connect channels to types. In other words, we cannot use the SDL inheritance for inheriting communication. Therefore we have to model our own model for inheriting "communication" associations.

The inheritance of associations is done on UML level as part of the pre-processing. There are two algorithms to process inheritance of associations: conservative scenario and full-connect scenario. The models shown in Figure V-5 are preprocessed in both ways to illustrates the algorithms. The resulting models are shown in Figure IV-6 and Figure IV-7.

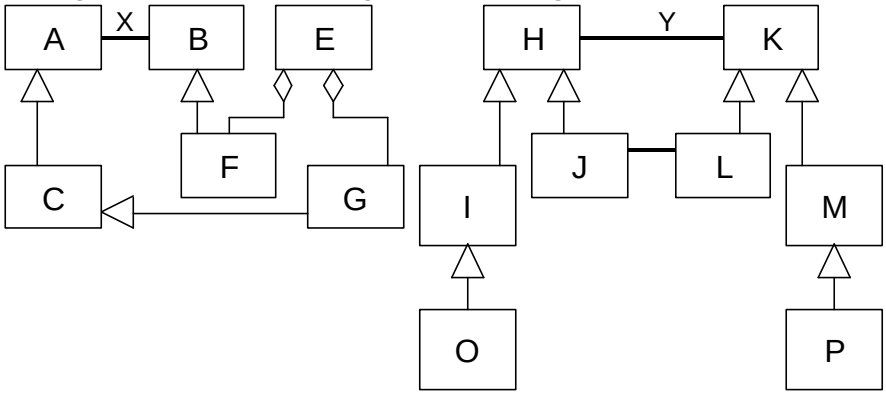

Figure V-5. Example of association before resolving inheritance

In the **conservative** approach, an association between two classes A and B is inherited by taking a subclass of A and a subclass of B at the same time. The original association is then copied between the two subclasses and between all other combinations of a subclass on both sides. This is repeated for the next level of inheritance, until there are not any subclasses left on both sides.

Preconditions	– *communication option* = conservative – either *assciation.fromclass* or *association.toclass*, or both, have at least one

	subclass (inheritance)
Context	– *fromclass* = *association.fromclass* – *toclass* = *association.toclass* – sub_{from} = {c ∈ *package.classes* | *c.superclass* = *fromclass*} – If sub_{from} = ∅, let sub_{from} = {*fromclass*} – sub_{to} = {c ∈ *package.classes* | *c.superclass* = *toclass*} – If sub_{to} = ∅, let sub_{to} = {*fromclass*}
Action	For all (*subfrom*, *subto*) ∈ {(*from*, *to*) | *from*∈ sub_{from} , *to* ∈ sub_{to}} do – Let *subassoc* be an exact copy of *association* – *subassoc.fromclass* = *subfrom* – *subassoc.toclass* = *subto* – If *subfrom* = *fromclass* then *subassoc.name* = *subassoc.name* + "_" + *subto.name*, else if *subto* = *fromto* then *subassoc.name* = *subassoc.name* + "_" + *subfrom.name*, else *subassoc.name* = *subassoc.name* + "_" *subfrom.name* + "_" + *subto.name* – Add *subassoc* to *package.associations* – Apply this rule recursively to *subassoc*

Rule 22. Conservative Association Inheritance

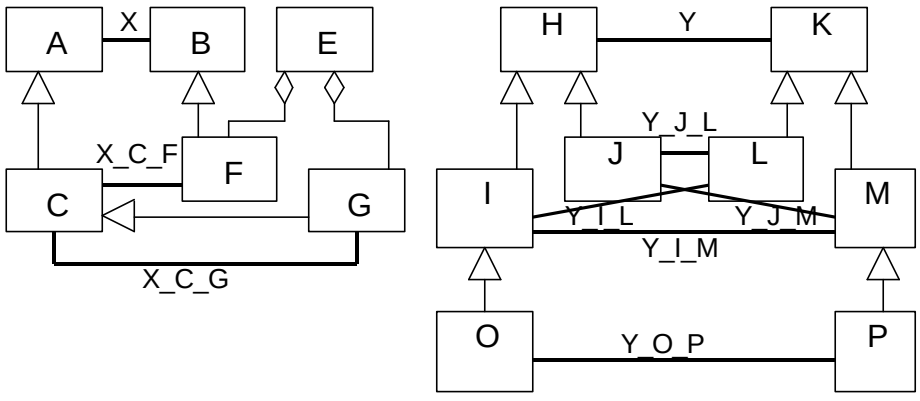

Figure V-6. Example after Conservative Association Inheritance

In the **full connect** scenario, we create a copy of an association between any combination of subclasses of both end of the association.

Preconditions	– *association.communication* = full – either *assciation.fromclass* or *association.toclass*, or both, have at least one subclass (inheritance)

Context	– *fromclass = association.fromclass* – *toclass = association.toclass* – sub_{from} = {c ∈ *package.classes* | ∃ (c_1,…,c_n) : ∀i∈(1..n): c_i ∈ *package.classes*, ∀i∈(1..n-1) c_{i+1}.*superclass* = c_i, c_1 = *fromclass*, c_n = c} – sub_{from} = {c ∈ *package.classes* | ∃ (c_1,…,c_n) : ∀i∈(1..n): c_i ∈ *package.classes*, ∀i∈(1..n-1) c_{i+1}.*superclass* = c_i, c_1 = *fromclass*, c_n = c}
Action	For all (*subfrom*, *subto*) ∈ {(*from*, *to*) | *from*∈ sub_{from} , *to* ∈ sub_{to}} do – Let *subassoc* be a exact copy of *association* – *subassoc.fromclass = subfrom* – *subassoc.toclass = subto* – If *subfrom* = *fromclass* then *subassoc.name* = *subassoc.name* + "_" + *subto.name*, else if *subto* = *fromto* then *subassoc.name* = *subassoc.name* + "_" + *subfrom.name*, else *subassoc.name* = *subassoc.name* + "_" *subfrom.name* + "_" + *subto.name* – Add *subassoc* to *package.associations*

Rule 23. Full Connect Association Inheritance

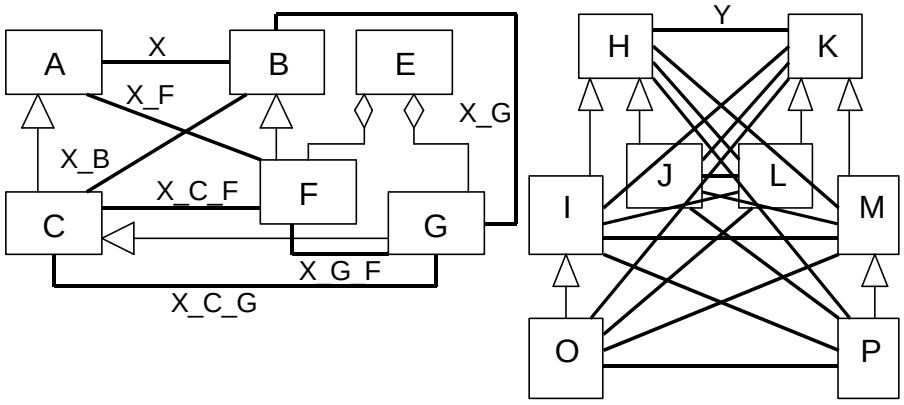

Figure V-7. Example after Full Connect Inheritance of Association

V.2.3.6 Preprocessing of State Diagram

A state diagram is a collection of states and transitions. Most of the preprocessing is performed on state level, therefore there is only one rule in this section. Rule 29 makes sure that there is only one start transition. If there are two transitions leaving from a top-level start state, it only uses the first transition and deletes the other.

Context	– *statediagram* is a state diagram – ($tran_1$, …, $tran_m$) = *statediagram.transitions*

Preconditions	– $\exists$ i,j : i $\neq$ j $\in$ (1..m): $trans_i$.source.type = start, $trans_j$.source.type = start, $trans_i$.source.superState = empty and $trans_j$.source.superState = empty
Action	Delete $trans_j$. Repeat this rule.

Rule 24. Only one start transition

V.2.3.7 Preprocessing of State

Context for this section	– *state* is a state – *statediagram* is a state diagram containing state

Rule 25. Context for preprocessing state

The following two rules define some functions concerning substate diagrams. These functions are used for the preprocessing of transitions.

Function	*substates(state)* = $\{s \in statediagram.states \mid \exists (s_1,\ldots s_n) : s_1 = s \wedge \forall\ i \in 1..n\text{-}1: s_i = s_{i+1}.superstate\}$

Rule 26. Function definition for substates.

Function	*substartstates(state)* = $\{s \in statediagram.states \mid s.superstate = state \wedge \exists\ t \in statediagram.transitions: t.source.type$ = start $\wedge\ t.dest = s\}$ If *substartstates(state)* = $\varnothing$, *substartstate(state)* = empty, else *substartstate(state)* = any element from *substartstates(state)*

Rule 27. Function definition for start state of substate diagram

V.2.3.8 Preprocessing of Transition

Context	– *transition* is a transition – *dest = transition.dest* – *source = transition.source*

Rule 28. Context for Preprocessing Transition

The most difficult part in preprocessing the state diagrams is flattening substates in a hierarchical state diagram. The substates need a copy of the transitions of their superstates, but these transitions need to be expanded with additional exit actions for the substate. Also, the destination of a transition should be changed to the initial state of the substate diagram of the destination state. Figure V-8 shows an example of a state diagram with with two superstate and three substates. Many states and transitions contain actions to illustrate the algorithm. Figure V-9 shows the same state diagram after the preprocessing.

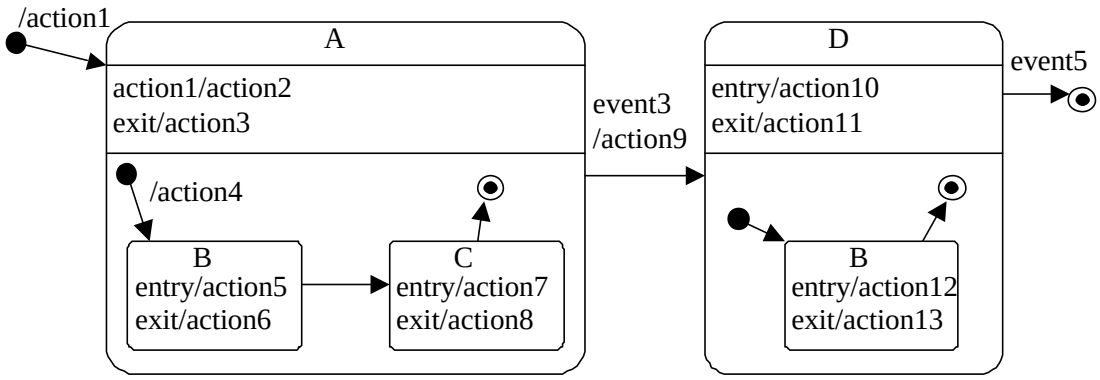

Figure V-8. Example of Nested State Diagram

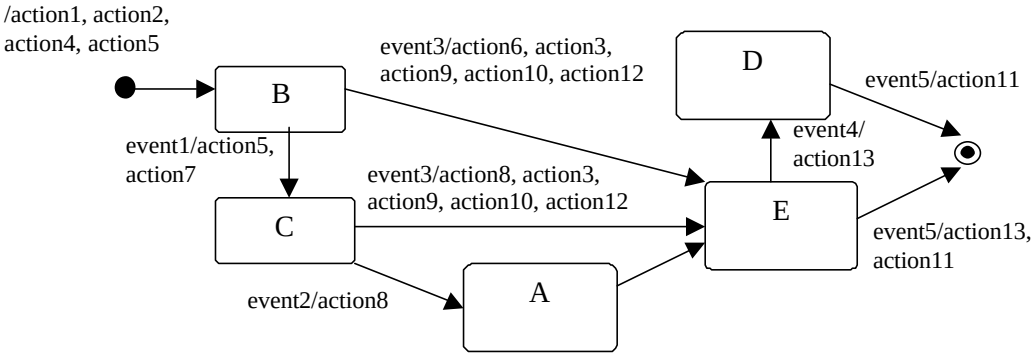

Figure V-9. Flatened version of Nested State Diagram

The rules below process one transition at a time. Rule 29 collect the entry and exit actions of the state that are crossed by the transition. In our example, none of transitions crosses state boundries, so this rule has no effect on our example. Rule 30 appends the exit actions of the source state to the beginning of the transition. Rule 31 and Rule 32 are executed alternately to append the entry actions of the destinations state and to look for a possible sub start state. These two rules are repeated until there is no sub start state. Rule 33 and Rule 34, finaly, finds out the new destination state. The "+" operator used in the translation rules below append the list of actions.

Preconditions	– *source.supertype* ≠ *dest.superstate*
Context	– Let *super* be the common superstate for *source* and *dest*, or empty if the state diagram is the common super state. – Let $(s_1,\ldots,s_n)$: s_1 = *super*, s_n = *source*, $\forall i \in (2..n)$: $s_{i-1} = s_i.superstate$ – Let $(d_1,\ldots,d_m)$: s_1 = *super*, d_m = *dest*, , $\forall i \in (2..m)$: $d_{i-1} = d_i.superstate$
Action	– $transition.actions = s_{n-1}.exitactions + \ldots + s_2.exitactions + transition.actions + d_2.entryactions + \ldots + d_{m-1}.entryactions$

Rule 29. Collect exit and entry actions of superstates.

Action	– *transition.actions* = *transition.source.exitactions* + *transitions.actions*

Rule 30. Append exit actions of source state

Action	– *transition.actions* = *transitions.actions* + *transition.dest.entryactions*

Rule 31. Append entry actions of destination state

Preconditions	– *substartstate(dest)* ≠ empty
Context	– *startstate* = *substartstate(dest)* – Let *t* ∈ *statediagram.transitions*, where *t.source.type* = start ∧ *t.dest* = *startstate.*
Action	– *transition.actions* = *transition.actions* + *t.actions* – *transition.dest* = *startstate* – repeat from Rule 31.

Rule 32. Duplicate transitions to substates

Preconditions	– *dest.type* = stop – *source.supertype* ≠ empty – *source.superstate* = *dest.superstate*
Action	– *transition.dest* = *source.superstate*

Rule 33. Move terminal transitions to superstate.

Preconditions	– *source.substates*() ≠ ∅
Action	For each *substate* ∈ { s ∈ *source.substates*() | *s.type* = normal } : – create a copy of *transition*, called *transcopy* – *transcopy.source* = *substate* – *transcopy.actions* = *substate.exitactions* + *transcopy.actions* – repeat this rule (Rule 34) with *source* = *transcopy.source*

Rule 34. Duplicate transitions to substates

V. 3 SDL Information Model

The SDL information model provides the means to store and manipulate an SDL specification. Many of the translation rules refer to this information model to access information in or to make changes to the SDL specification. The SDL information model must be able to contain all the details of the SDL language, even those aspects that does not have a mapping with UML. The reason is of course that, after the round-trip engineering, the SDL model is exported to become the new version of the SDL specification. Information not stored in the model is lost.

The SDL information model is designed for easy access and manipulation of the SDL entities involved in the mapping. Figure V-9 shows the generalization structure of the complete SDL information model. A number of generalizations allow us to reason on a more abstract level. For example, systems, blocks and processes are all structures that may contain declarations, communication or other structures. In addition, of all possible declarations possible in SDL, only those declarations that have a mapping to SDL are made explicit. Some of the declaration that have no explicit representation in our model are: syntype, timer, synonym and select.

The presented information model is *not* a general purpose data structure. Only the entities relevant for the round-trip engineering with UML are explicitly modeled. When parsing an SDL specification, the information not modeled in the information model is stored invisibly to allow the export of a full SDL specification after a translation. The extra information includes graphical position information.

V 3.1 Entity Inheritance Hierarchy

The SDL information model is structured as an inheritance hierarchy, see Figure V-9. The translation rules often uses a super type to refer to any kind of subtype. For example, a list of *communications* may contain channels and/or signal routes and a *structure* pointer can be filled in with a system, package, (typed) block or (typed) process. This way of working makes the information model more compact and the translation rules easier to write.

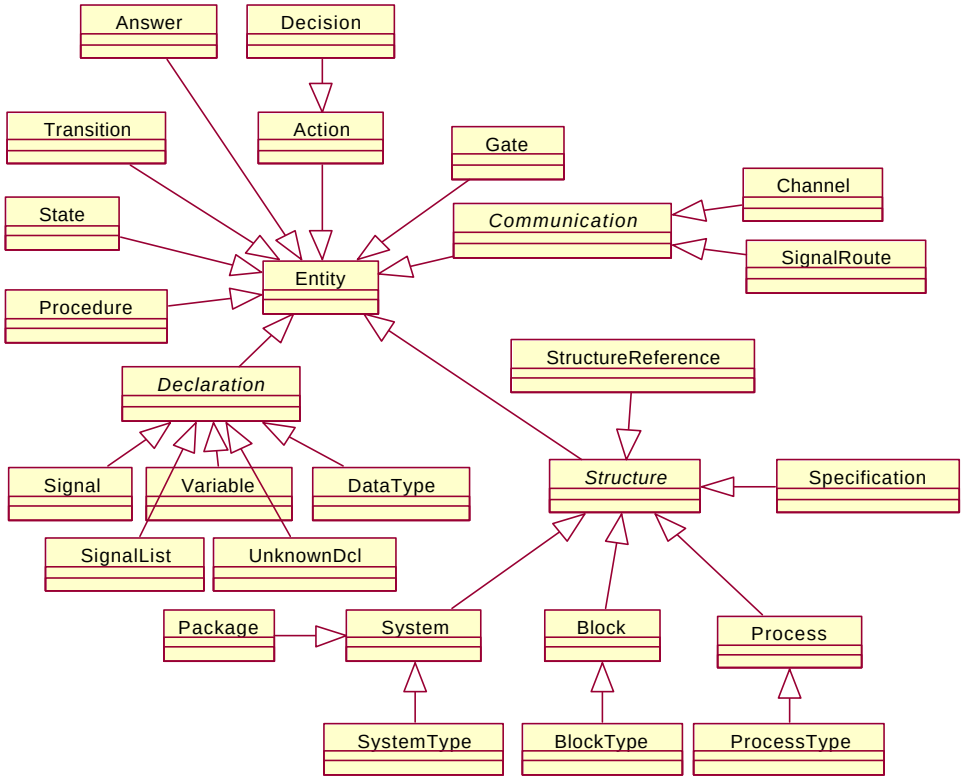

Figure V-9. Inheritance Structure of SDL Entities

In the tables below, we define the exact information model for SDL that is used to apply the incremental translation rules. The super type of an entity is denoted between angle brackets (<>), meaning that the entity inherits all the attributes of the super type. Note that more information, i.e. more attributes, is necessary to contain a complete SDL parse tree. This extra information is hidden for our purpose and is only needed to write back the finished SDL specification at the end of the iteration.

The first type, SDL entity, is the super type for all other SDL types. Besides the name and comment attribute, it provides a unique identification to all entities, which are used internally to set up the UML-SDL link as described in section V. 4.

SDL Entity	Description
name	Name of the entity
comment	Comment attached to the entity
id	Identifier used internally for the UML-SDL link

Table V-14. Common Attributes for all SDL entities

V 3.2 Static Structure

SDL structure holds the common attributes for SDL system, package, block (type) and process (type). It allows us to reason on any kind of structure in a uniform way, e.g. to create a communication route between two structures, independent of their concrete type. The *parent* and

system attributes are used internally to conveniently access its relatives. The attributes *declarations*, *children* and *communication* contain the actual contents of the structure.

UML Structure	Description
<entity>	Inherit the attributes of SDL entity
formal parameters	String
parent	Reference to the parent structure
system	Reference to the parent system or package
declarations	List of declarations
children	List of structures
communications	List of channels and signal routes in this structure

Table V-15. Common Attributes of all SDL structures

SDL Specification	Description
<structure>	Inherit the attributes of SDL structure

Table V-16. Attributes of SDL specification

SDL Block	Description
<structure>	Inherit the attributes of SDL structure
block type	Is empty for regular blocks (block definition). For block instances, it refers to the block type the instance is based on.

Table V-17. Attributes of SDL Block

SDL Block Type	Description
<block>	Inherit the attributes of SDL block
virtuality	String, defines the ability to subtype
specialization	Reference to the super-block type
gates	List of gates

Table V-18. Attributes of SDL Block Type

SDL Process	Description
<structure>	Inherit the attributes of SDL structure
process type	Is empty for regular process (process definition). For process instances, it refers to the process type the instance is based on.
number instances	String without brackets, e.g. 1,4
start	The start transition
states	List of states

Table V-19. Attributes of SDL Process

SDL Process Type	Description
<process>	Inherit the attributes of SDL process
virtuality	String, defines the ability to subtype
specialization	Reference to the super-structure
gates	List of gates

Table V-20. Attributes of SDL Process Type

SDL Procedure	Description
<structure>	Inherit the attributes of SDL process (procedure also contains a state diagram)
parameters	List of parameters
returns	Sort (String)

Table V-21. Attributes of SDL Procedure

SDL Parameter	Description
<entity>	Inherit the attributes of SDL entity
variable	String
type	Sort (String)

Table V-22. Attributes of SDL Parameter

V 3.3 Communication

Communication is our common notion of channels and signal routes. It allows us to reason on communication routes independently of what they connect (processes or blocks). SDL channel and signal route are concrete subentities of communication, but do not have any extra attributes. That reflects the facts that channels and signal routes only differs semantically (delaying, non-delaying) and not syntaxtically.

SDL Communication	Description
<entity>	Inherit the attributes of SDL entity
bidirect	True if communication is in both directions
from struct	Reference to the “from” structure of the communication. Equals “ENV” if the communication comes from the environment.
to struct	Reference to the “to” structure of the communication. Equals “ENV” if the communication goes to the environment.
from connect	Reference to the gate or channel the “from” side is connected with.
to connect	Reference to the gate or channel the “to” side is connected with.
from to signal list	List of signals and signal lists on the communication going from “from” to “to”.
to from signal	List of signals and signal lists on the communication going from “to” to “from”.

Table V-23. Attributes of SDL Communication

SDL Channel	Description
<communication>	Inherit the attributes of SDL communication

Table V-24. Attributes of SDL Channel

SDL Signal Route	Description
<communication>	Inherit the attributes of SDL communication

Table V-25. Attributes of SDL Signal Route

SDL Gate	Description

<entity>	Inherit the attributes of SDL entity
bidirect	True if communication is in both directions
to constraint	Identifier string that refers to a type specification.
from constraint	Identifier string that refers to a type specification.
out signal list	List of signals and signal lists on the communication going the type.
in signal list	List of signals and signal lists on the communication going into the type.

Table V-26. Attributes of SDL Gate

V 3.4 Declarations

We use declaration as a super type for everything that is specified in an SDL textbox. We explicitly define subtypes for the kind of declaration we need for the translation. Other kinds of declarations are stored as *Unknown Dcl's.* Note that the attribute *name* is inherited from the *entity* super type.

SDL Declaration	**Description**
<entity>	Inherit the attributes of SDL entity
declared in	Reference to the structure where the declaration is defined.

Table V-27. Attributes of SDL Declaration

SDL Signal	**Description**
<declaration>	Inherit the attributes of SDL declaration
parameters	List of strings, the "sort" of the parameters.

Table V-28. Attributes of SDL Signal

SDL SignalList	**Description**
<declaration>	Inherit the attributes of SDL declaration
signals	List of signals contained in the signal list.
signal lists	List of signal lists contained in the signal list.

Table V-29. Attributes of SDL Signal List

SDL Datatype	**Description**
<declaration>	Inherit the attributes of SDL declaration
signature	String containing the data part of the data type. A list of attributes for a "struct" newtype.
behaviour	String containing the operator part of the data type. A list of operators for a "struct" newtype.

Table V-30. Attributes of SDL Datatype

SDL Variable	**Description**
<declaration>	Inherit the attributes of SDL declaration
type	The sort of the variable.
initial expr	The initial value of the variable.

Table V-31. Attributes of SDL Signal

SDL Unknown Dcl	**Description**
<declaration>	Inherit the attributes of SDL declaration

Table V-32. Attributes of SDL Unknown Dcl

V 3.5 State Machine

The process structure holds two attributes that represents the state machine: *start* and *states*. *Start* points to the initial start transition of the state machine. *States* is the list of states contained in the FSM. Each state contains the list of transitions that leave from that state.

SDL State	Description
<entity>	
saves	List of strings
transitions	List of SDL Transitions

Table V-33. Attributes of SDL State

SDL Transition	Description
<entity>	
virtuality	String stating the virtuality contraint
input	String representing of the signal
enable	String of the enabling condition
priority	Boolean
save	Boolean
actions	List of SDL actions
start	bool

Table V-34. Attributes of SDL Transition

SDL Action	Description
<entity>	
action	String (e.g. output, task, call, nextstate, stop)
body	String containing the rest of action, may be empty

Table V-35. Attributes of SDL State

SDL Decision	Description
<action>	

Table V-36. Attributes of SDL State

SDL Answer	Description
<entity>	

Table V-37. Attributes of SDL State

V. 4 Link UML and SDL

V 4.1 Hierarchical Links

As already explained in section IV 3.5, we need a link between the UML model and the SDL specification. During the translation, we maintain hierarchical links between the models and the specification. Each entity in the UML model is linked with its corresponding SDL entities. For example, a «block» class is, among others, linked with the generated block and management process. An association is linked with all generated signal routes and channels. Figure V-10 shows an limited overview of the UML and SDL information models and the links between the corresponding constructs. An example of the links between two concrete models is shown in Figure IV-13.

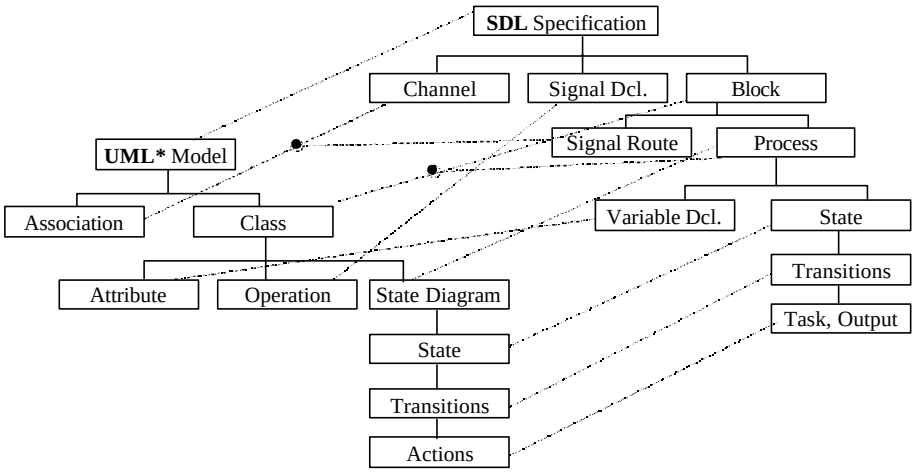

Figure V-10: Hierarchical Links between UML and SDL

In order to realize this link, we extend the UML information model defined in section V. 2 with links to the SDL information model defined in V. 2. These links forms a 1-to-n relationship between UML entities and SDL entities. For example, an association is linked with many channels and signal routes, but a specific channel is the result of exactly one association. This chararacteristic provides the possibility to use the UML-SDL in the reverse direction, see section V 4.3 for details.

V 4.2 UML link extension

We extend the UML information model defined in V 2.1 with links to the SDL data structure. These links enables us to translate changes in a particular context. For example, if a new attribute has been added to a class, then a variable declaration should be added to the process linked with the class. As one can see below, most UML entities have more than one link to SDL. Although

some of the links may be redundant in some cases, all links are necessary to allow correct round-trip engineering.

Note that we use unique identifiers as the link to SDL entities. In this way, a UML entity can point to the same entity in different SDL specifications at the same time. This aspect is important when using the reverse link during reverse iteration. An additional advantage is that the link can easily be stored in file.

Initially the links are constructed during the translation of new UML entities. For example, if a management process is generated from a class, the generated process is assigned to the *sdlprocess* attribute of the class. The next time, during synchronization, the UML and SDL data structures are both build-up and the links created during the translation are restored. Without storing explicit link information in UML or SDL files, it is difficult to restore the original links.

Package SDL link	**SDL Type**	**Info**
sdlspecification	Specification	Link to the complete SDL specification.

Table V-38. SDL links of UML model

Package SDL link	**SDL Type**	**Info**
sdlspecification	Specification	Back link to the complete SDL specification.
sdlsystem	System (Type) or Package	The system or package that is linked with this package.
sdlarchitecture	System (Type) or Block	Link with the structure that holds the instances of top-level classes.

Table V-39. SDL links of UML package

Class SDL link	**SDL Type**	**Info**
sdldefinition	Block(Type) or Process(Type)	Link with the main structure generated from the class.
sdlsuper	Block Type or Process Type	The type definition which is linked with the superclass.
processesblock	Block	Link with the extra block containing the processes. May be empty or may be the same as sdldefinition.
declarationStruct	Structure	Pointer to the structure that contains the declaration generated by the class
sdlprocess	Process	Link with the process generated from the class. For «process» classes, sdldefinition = sdlprocess.
sdlsignallist	Signallist Declaration	
sdldatatype	Newtype Declaration	Only applicable for «newtype» classes.

Table V-40. SDL links of UML class

Operation SDL link	**SDL Type**	**Info**
sdlsignal	Signal Declaration	Link to the signal declaration. Only applicable for

		«signal» operations.
sdlprocedure	Procedure Definition	Link to the procedure definition. Only applicable for «procedure» operations.

Table V-41. SDL links of UML Operation

Attribute SDL link	SDL Type	Info
sdldeclaration	Variable Declaration	Link to the variable declaration generated from the attribute.

Table V-42. SDL links of UML Attribute

Aggregation links	SDL Type	Info
sdlcomponent	Structure	Link with the structure definition or the type based instance.
sdldeclaration	Variable Declaration	Link to the variable declaration generated from the aggregation.

Table V-43. SDL links of UML Aggregation

The UML association has a complicated mapping; therefore, it also needs many links to keep track of all the SDL entities that are generated from an association. The links used for a particular association depends heavily on the option settings and on the classes it is connected with. The most complicated case is where a full-communication association connects two processes in a completely different scope.

Association links	SDL Type	Info
sdlfromroute	Signal Route	Link to the signal route generated from the “From” association end. May be empty.
sdltoroute	Signal Route	Link to the signal route generated from the “To” association end. May be empty and may be the same as sdlfromroute.
sdlfromchannel	Channel	Link to the main channel generated from the “From” association end. May be empty.
sdltochannel	Channel	Link to the main channel generated from the “To” association end. May be empty and may the same as sdlfromchannel.
sdlchannels	List of Channels	A list of links to all the channels generated from the association, including the sdlfromchannel and sdltochannel.
sdlgates	List of Gate	A list of links to all the gates generated from the association.
fromsignallist	Signallist	Link to the SDL signallist that contains the signals that can be send to the From association end. May be the same as *fromclass.sdlsignallist*.
tosignallist	Signallist	Link to the SDL signallist that contains the signals that can be send to the To association end. May be the same as *toclass.sdlsignallist*.

Table V-44. SDL links of UML Association

State Diagram links	SDL Type	Info
sdlprocess	Process(Type)	Link with the process or process type definition.

Table V-45. SDL links of UML State Diagram

State links	SDL Type	Info
sdlstate	State	Link with the SDL state.

Table V-46. SDL links of UML State

Transition links	SDL Type	Info
sdltransition	Transition	Link with the SDL transition.
timerdeclaration	Declaration	
nextstate	Action	Link with the nextstate statement, which corresponds with the destination state.

Table V-47. SDL links of UML Transition

Action links	SDL Type	Info
sdlaction	Transition	Link with the SDL transition.

Table V-48. SDL links of UML Action

V 4.3 SDL ADT extension

To write down the reverse iteration translation rules, we need links from SDL to UML. However, we define these links indirectly in terms of the UML-SDL links. As explained before, the UML-SDL links can also applied in the reverse direction. If *class.sdlprocess* results in a certain *process*, then the reverse funtion *process.sdlprocess*$^{-1}$ results in the original *class*. In other words, *process.sdlprocess*$^{-1}$ must be interpreted as the function that returns the class that is linked with *process* with the *sdlprocess* link. For almost every link defined in the previous section, we define a reverse link.

SDL Type	Reverse Link	Return Value
Specification	*sdlspecification*$^{-1}$	The model that links with this specification.

Table V-49. Reverse Links resulting in a UML model

SDL Type	Reverse Link	Return Value
System (Type) or Package	*sdlsystem*$^{-1}$	The package that links with this system or package as the *sdlsystem*.
System (Type) or Block	*sdlarchitecture*$^{-1}$	The package that uses this structure (system or block) as the architecture block.

Table V-50. Reverse Links resulting in a UML Package

SDL Type	Reverse Link	Return Value
Block(Type) or Process(Type)	*sdldefinition*$^{-1}$	The class that links to this block or process as its main structure.
Block Type or Process Type	*sdlsuper*$^{-1}$	The class that links to this structure as its supertype.
Block	*processesblock*$^{-1}$	The class that links to this block as its processes block.
Process	*sdlprocess*$^{-1}$	The class that links to this process as its process as a main structure or as its management process.

Signallist Declaration	*sdlsignallist*$^{-1}$	The class that has generated this signal list.
Newtype Declaration	*sdldatatype*$^{-1}$	The class that has generated this declaration.

Table V-51. Reverse Links resulting in a UML Class

SDL Type	Reverse Link	Info
Signal Declaration	*sdlsignal*$^{-1}$	The operation that generated this signal.
Procedure Definition	*sdlprocedure*$^{-1}$	The operation that generated this procedure.

Table V-52. Reverse Links resulting in a UML Operation

SDL Type	Reverse Link	Info
Variable Declaration	*sdldeclaration*$^{-1}$	The attribute that generated this variable.

Table V-53. Reverse Links resulting in a UML Attribute

SDL Type	Reverse Link	Info
Structure	*sdlcomponent*$^{-1}$	The aggregation that generated this structure definition or instance.
Variable Declaration	*Pid*$^{-1}$	The aggregation that generated this Pid variable declaration.

Table V-54. Reverse Links resulting in a UML Aggregation

SDL Type	Reverse Link	Info
Signal Route / Channel	*sdlfrom*$^{-1}$	Association that generated this channel or signal route from the “From” association end.
Signal Route / Channel	*sdlto*$^{-1}$	Association that generated this channel or signal route from the “To” association end.
Signal Route / Channel	*sdlchannel*$^{-1}$	Association that generated this channel or signal route.
Gate	*sdlgate*$^{-1}$	Association that generated this gate.

Table V-55. Reverse Links resulting in a UML Association

SDL Type	Reverse Link	Info
State	*sdlstate*$^{-1}$	The UML state that generated this state.

Table V-56. Reverse Links resulting in a UML State

SDL Type	Reverse Link	Info
Transition	*sdltransition*$^{-1}$	The UML transition that generated this state.
Action	*nextstate*$^{-1}$	The UML transition

Table V-57. Reverse Links resulting in a UML Transition

V. 5 Compare & Translate

After loading and preprocessing the UML model(s) and SDL specification(s), the next step is comparing the two UML models (forward iteration) or the two SDL specifications. The comparison is performed in a hierarchical way and is based on identifiers instead of names in order to detect renames. An entity that is present in the old model but not in the new model is considered *deleted*. An entity that is not present in the old model but is present in the new model is *new*. An entity present in both models is said to *match* and is compared to find differences in its attributes. For each change or match, the corresponding set of translation rule is processed. For example, if a class is found in the old and the new UML model with the same identifier, then all the rules in section V 6.10 are processed one by one. If, for instance, the names of the two classes are different, then either Rule 73 or Rule 74 will effectively be executed, depending on the stereotype of the class.

The order in which the different constructs are compared is not arbitrary. For example, a new attribute cannot be translated before its class has been translated and an association should be deleted before the classes it is connected with are deleted. The tree shown in Figure V-11 defines the order in which the various parts of the UML model are compared and processed. The tree is incomplete in the sense that the sub trees of the *new* and *delete* changes are left out of the picture. It is clear, that in a *new* package, all its classes, aggregations and associations are also new. When translating a *new* class, all its attributes and operations are also translated as *new*. Similarly, when deleting a class, its attributes and operations are deleted first.

As a general rule, we first translate deleted entities, then compare matching items and then translate new items. When translating deleted classes, the leave components in the aggregation tree are translated first, then their aggregates and so on. New classes are processed in the opposite order, i.e. aggregates first. Note that the translation rules in the next sections are not ordered in order of execution, but are grouped per entity in the orther *new*, *delete* and *compare*.

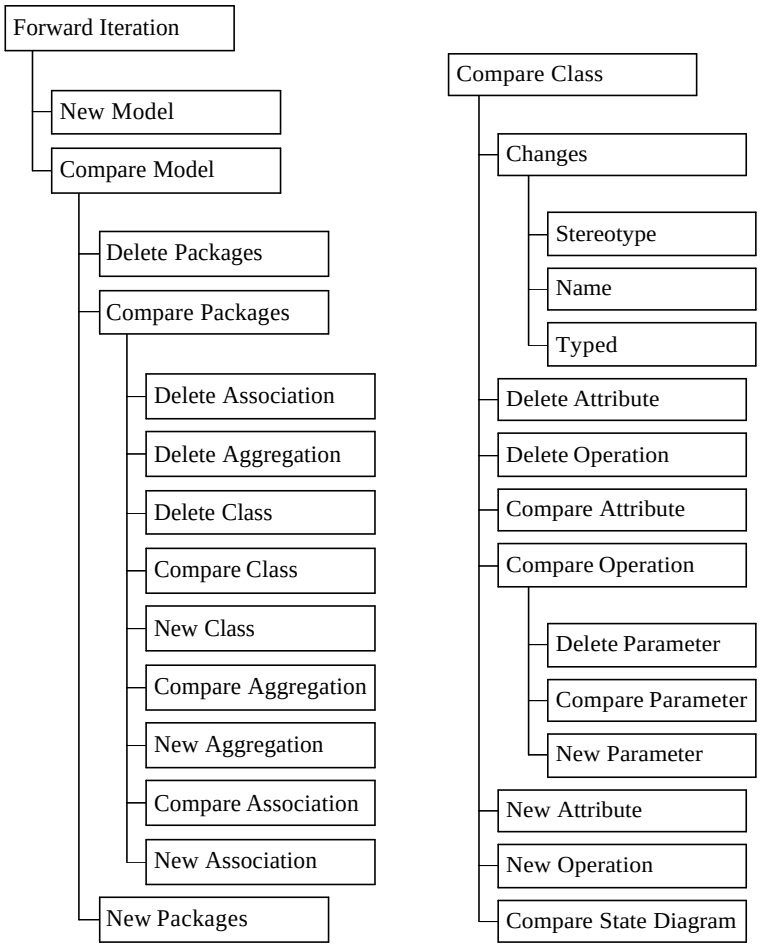

Figure V-11. Hierarchy and Order of Model Comparison

V. 6 UML to SDL

V 6.1 Introduction

This section contains all rules necessary to translate any change in a UML model into a modification of the SDL specification. The rules are grouped in rule sets, i.e. each section is a rule set that translates a change. The *compare* sections compare the attribute of the entity and only translate something if a change is detected. The *new* and *delete* sections always translate the change, but uses other information (stereotype, relationships, UML-SDL links, etc.) to find out exactly which rules need to be fired. Each rule set is independent of the other rules sets, therefore each section starts by defining a list of context variables that are used in the rules.

V 6.2 New Model

The first rule defines the translation of a new UML model. For reasons of uniformity, this rule is also described as a change, i.e. the UML model is new as compared with nothing. This rule is actually the starting point to translate a new UML model for the first time, i.e. when there is nothing to compare the model with yet.

A UML model maps on an SDL specification. Therefore, for a new *model* we create a new empty SDL specification called *spec* with the same name. The names of the model and the specification actually represent the filenames and are not really part of the model or specification.

Context	– *model* is the new UML model
Action	Create an empty SDL specification = *spec*. Set *spec.name* = *model.name*
Variables	– *model.sdlspecification* = *spec*

Rule 35. Translate Change Package Name

Note that after executing this rule, the packages, classes and other information contained in the model are all considered “new” and are all translated with their respective translation rules. The order of execution is globally defined in section V. 5 and is not repeated for each change.

V 6.3 Compare Model

The only information in a UML model that can change is the name. The other entities contained in the model are compared separately. Again, the order of execution of comparison is defined globally in section V. 5.

Context	– $model_{old}$ is the old UML model – $model_{new}$ is the new UML model, being compared with $model_{old}$
Preconditions	– $model_{old}.name \neq model_{new}.name$
Action	$model_{new}.\underline{sdlspecification}.name = model_{new}.name$

Rule 36. Change Model Name

V 6.4 Delete Model

Deleting a UML model corresponds to deleting all available information. For safety reasons, this rule should only be executed after user confirmation. This rule has little practical value and is presented here for the matter of completeness.

Context	– *model* is the old UML model
Action	Delete *model.*$\underline{sdlspecification}$

Rule 37. Translate Deleted Model

V 6.5 New Package

Packages are UML's main structuring mechanism. They group classes that belong together, although classes from different packages can be placed next to each other in one diagram and connected with a relationship. Packages can be nested or connected with a dependency relationship. In SDL, the package concept is very similar, but the contents of SDL packages are strictly separated. An SDL package can contain all kind of declarations, e.g. signal, data types, block types and process types, but not block or process definitions. A system that imports such a package may use these declarations, e.g. to define process instances, but it may not directly refer to it like in UML. An SDL specification usually contains one system definition and many packages with a hierarchy of uses relationships.

When translating a UML model for the first time, this rule set (New Package) is fired for the main package and for all the nested packages in the model. When comparing two models, this rule set is fired only for completely new packages.

Note that before executing the rules below, the UML package has already been preprocessed by the algorithm defined in V.2.3.1. Therefore, the system class is already defined as the class that represents the package.

Context	– *model* is the UML model – *package* is the new UML package – *sysclass* = *package.systemclass* – $\underline{spec}$ = *model.*$\underline{sdlspecification}$

Variables	– *package.sdlspecification* = *spec*

Rule 38. Translate Context for New Package

A UML package is translated into SDL as a system or a package, depending on the stereotype of the package, see Rule 39 and Rule 41. If the stereotype is «system» and the system class is typed, the system is generated as a combination of a system type and a type based system instance, see Rule 40. An important aspect of these translation rules is the assignment of the *architecture* and *sdldefinition* variables. They respectively define where top-level instances and top-level types should be placed. As an SDL package cannot contain instances, Rule 41 creates an extra block type to hold the architecture (an instance of each top-level class).

Preconditions	– *package.stereotype* = «system» – *sysclass.typed* = false

Action	Add to *spec*:
	`SYSTEM <package.name>;`
	`ENDSYSTEM <package.name>;`

Variables	– *package.sdlsystem* = *system* = the added system – *package.architecture* = *system* – *sysclass.sdldefinition* = *architecture*

Rule 39. Translate non-typed «system» Package

Preconditions	– *package.stereotype* = «system» – *sysclass.typed* = true

Action	Add to *spec*:
	`SYSTEM TYPE <package.name>;`
	`ENDSYSTEM TYPE <package.name>;`
	`SYSTEM a_<package.name> : <package.name>;`
	Translate *package.systemClass* as a new class, starting from Rule 55.

Variables	– *package.sdlsystem* = *system* = the added system type – *package.architecture* = *system* – *sysclass.sdldefinition* = *architecture*

Rule 40. Translate Typed «system» Package

Preconditions	– *package.stereotype* = «package»

Action	Add to *spec*:
	`PACKAGE <package.name>;`
	`  BLOCK TYPE <package.name>;`
	`  ENDBLOCK TYPE <package.name>;`

	`ENDPACKAGE <package.name>;` Translate *package.systemClass* as a new class, starting from Rule 55.
Variables	– *package.sdlsystem* = *system* = the added package – *package.architecture* = the block type in the package – *sysclass.sdldefinition* = *architecture*

Rule 41. Translate «package» Package

Rule 43 uses the "new class" translation rules on the system class to create a processes block and management process at system level. The definition structure of the system class has already been created in the previous rules.

Action	Translate *package.systemClass* as a new class, starting from Rule 55.

Rule 42. Create Processes Block and Management Process

Context	– *package.packages* = (p_1, …, p_n)

Action	Add to *package.sdlsystem*:
	
	`USE <`p_1`.name>;`
	…
	`USE <`p_n`.name>;`

Rule 43. Translate Package Dependencies

V 6.6 Delete Package

Deleting a package in UML simply results in deleting the SDL system or package linked with the UML package. In case that the different entities are stored in separated files, only the reference in the specification to the system or package should be deleted and not the files themselves. Even so, the execution of this rule should preferable be confirmed by the user, because it has big implications.

Preconditions	– Package *package* is **deleted**.
Context	– *system* = *package.sdlsystem* – *architecture* = *system* or the block type in *package*
Action	Delete *package.sdlsystem* from *spec*. If *sysclass.typed* = *true*, then first delete the instances of *package.sdlsystem* from *spec*.

Rule 44. Delete Package

V 6.7 Compare Package

In this rule set, only the name, stereotype and package dependencies are checked. The options (local/global) are compared for each class separately. For example, if the *global declaration* option changes from true to false, many signal declarations should be moved from system level to

a more local block or process. It is easier to make each class responsible for his signal declarations. This has the additional advantage that the local/global options can be set for each class separately.

Rule 45 defines many context variables to ease the specification of the translation rules. More important, however, is the initialization of the SDL links of the new model. Unlike the old model, the new model does not contain the restored SDL links. Each compare rule set therefore first copies the SDL links from the old entity to the new one.

Context for this section	– $model_{old}$ is the old UML model – $model_{new}$ is the new UML model, being compared with $model_{new}$ – $package_{old}$ is the previous UML package – $package_{new}$ is the new UML package to be compared with $package_{old}$ – $sysclass_{old} = package_{old}.systemclass$ – $sysclass_{new} = package_{new}.systemclass$
Variables	– $package_{new}.\underline{sdlsystem} = package_{old}.\underline{sdlsystem}$ *(set initial value)* – $package_{new}.\underline{architecture} = package_{old}.\underline{architecture}$ *(set initial value)*

Rule 45. Context for Comparing Packages

Preconditions	– $package_{old}.name \neq package_{new}.name$
Context	
Action	$package_{new}.\underline{sdlsystem}.name = package_{new}.name$ $package_{new}.\underline{architecture}.name = package_{new}.name$

Rule 46. Change Package Name

Rule 47 and Rule 48 defines how to translate a stereotype modification of a package. If the stereotype is changed from «system» to «package», an new architecture block is created and all non-typed structures and channels are moved into this block. In the other direction, the architecture block is simply eliminated after moving its contents to the system.

Preconditions	– $sysclass_{old}.stereotype$ = «system» – $sysclass_{new}.stereotype$ = «package» – $package_{new}.\underline{sdlsystem}$ is a system (type)
Action	Convert $package_{new}.\underline{sdlsystem}$ into a package and delete the system instance in the specification if present. Create an architecture block type in $package_{new}.\underline{sdlsystem}$ with name $package_{new}.name$ Move all block and process definitions, all type based instances and all channels from the system to the new architecture block type.
Variables	– $package_{new}.\underline{architecture}$ = newly created architecture block type

Rule 47. Change Package «system» to «package»

Preconditions	– $sysclass_{old}.stereotype$ = «package» – $sysclass_{new}.stereotype$ = «system» – $package_{new}.\underline{sdlsystem}$ is a package
Action	Change $package_{new}.\underline{sdlsystem}$ to a system. Move the complete contents of the architecture block to the system. Remove the architecture block.
Variables	– $package_{new}.\underline{architecture} = package_{new}.\underline{sdlsystem}$

Rule 48. Change Package «package» to «system»

Preconditions	– $\exists\ pack \in package_{old}.packages : pack \notin package_{new}.packages$
Action	Remove the clause "`use <pack.name>;`" from $package_{new}.\underline{sdlsystem}$

Rule 49. Remove Package Dependency

Preconditions	– $\exists\ pack \in package_{new}.packages : pack \notin package_{old}.packages$
Action	Add to the body of $package_{new}.\underline{sdlsystem}$: `USE <pack.name>;`

Rule 50. Add Package Dependency

V 6.8 New Class

As could be expected from an object oriented modeling technique, the *class* concept plays a very central role in UML. The extensibility mechanisms allow the class concept to be overloaded to mean different things. Similarly, there are many different ways of translating a class, depending on the value of the stereotype and translation options. Note that, because of the preprocessing, every class always has a stereotype and a default value for the options.

A class with stereotype «package» is translated as a package dependency, see Rule 52. A class with stereotype «block» is translated as a block or block type, see Rule 53 and Rule 54. If necessary, a processes-block and a management process are added to the block (type). From Rule 58 on, consecutively «process» classes, «newtype» classes and inheritance relationships are translated.

Context for this section	– *class* is the new UML class – *package* is the surrounding package of *class* – *sysclass = package.systemclass*

	– *system* = *package.sdlsystem* – *architecture* = *package.sdlarchitecture*

Rule 51. Translate Context for New Class

Preconditions	– *sysclass.stereotype* = «package»
Action	Add to *system*: `USE <class.name>;`

Rule 52. Translate Package Reference

Preconditions	– *class.stereotype* = «block» – *class.typed* = false
Context	– *definedIn* = *class.definedin.sdldefinition*

Action	Add to *definedIn*:
	`BLOCK <class.name>;`
	`ENDBLOCK <class.name>;`

Variables	– *structure* = the added block

Rule 53. Translate Non-typed «block» Class

Preconditions	– *class.stereotype* = «block» – *class.typed* = true
Context	– *definedIn* = *class.definedin.sdldefinition*

Action	Add to *definedIn*:
	`BLOCK TYPE <class.name>;`
	`ENDBLOCK TYPE <class.name>;`

Variables	– *structure* = the added block type

Rule 54. Translate Typed «block» Class

According to the SDL syntax rules, a block can either contain blocks or contain processes, not both at the same time. This is solved by creating an extra processes block, i.e. a block that contains all the processes that would otherwise be in the aggregate, see Rule 55. The *processesBlock* variable is assigned to the structure where the processes should be inserted. So, in Rule 56 the processes may be inserted in the block itself, as there are no «block» components.

Preconditions	– *class.stereotype* = «block» ∨ «system»∨ «package» – (*class.management* = true ∨ *class* has «process» components) ∧ (*class* has «block» components)
Context	– *structure* is the new block or process as defined before
Action	Add to *structure*: `BLOCK <class.name>_Processes;`

	`ENDBLOCK <class.name>_Processes;`
Variables	– *class.processesBlock* = the added block – *class.processesBlock* = *structure* if the last condition of the precondition is not fulfilled.

Rule 55. Provide Processes Block

Preconditions	– *class.stereotype* = «block» ∨ «system»∨ «package» – *class* has no components with stereotype «block»
Variables	– *class.processesBlock* = *structure*

Rule 56. No Processes Block Needed

A *management process* is a regular SDL process that fulfills the behavioral aspects of a block. Because of the SDL syntax rules, a block cannot contain variables or a state chart. When translating a «block» class with attributes or a state diagram, Rule 57 adds a management process to the block, which then will hold the generated variables and state chart. The *management* attribute of the class is set during the preprocessing.

Preconditions	– *class.management* = true – *sysclass.stereotype* = «block» ∨ «system» ∨ «package»
Context	– *structure* = *class.processesBlock*

Action	Add to *class.processesBlock*:
	`PROCESS <class.name>;`
	`ENDPROCESS <class.name>;`

Variables	– *class.sdlprocess* = the added process

Rule 57. Create Management Process

Rule 58 and Rule 59 create the process or process type for «process» classes. The *sdldefinition* and *sdlprocess* link variables of the class are both set to point to the generated process (type). The *sdldefinition* link is used to connect signal routes, gates and local signals. The *sdlprocess* link is used to generate the variables and the state chart. Rule 59 only generates the process type, not the instances. The type based instances of this process type are generated from the aggregations in Rule 80.

Preconditions	– *class.stereotype* = «process» – *class.typed* = false
Context	– *definedIn* = *class.definedin.sdldefinition*

Action	Add to *definedIn*:
	`PROCESS <class.name>;`
	`ENDPROCESS <class.name>;`

Variables	– *structure* = the added process – *class.sdldefinition* = *structure*

	– *class.*<u>*sdlprocess*</u> = <u>*structure*</u>

Rule 58. Translate «process» Class

Preconditions	– *class.stereotype* = «process» – *class.typed* = true
Context	– <u>*definedIn*</u> = *class.definedin.*<u>*sdldefinition*</u>

Action	Add to <u>*definedIn*</u>:
	`PROCESS TYPE <class.name>;`
	`ENDPROCESS TYPE <class.name>;`

Variables	– <u>*structure*</u> = the added process type – *class.*<u>*sdldefinition*</u> = <u>*structure*</u> – *class.*<u>*sdlprocess*</u> = <u>*structure*</u>

Rule 59. Translate Typed «process» Class

Rule 60 and Rule 61 fill in the link to the declaration structure (<u>*declarationStruct*</u>), depending on the *global declaration* option. Rule 62 uses this link to create the signal list associated with the class. The signals within the signal list are managed when translating operations.

Preconditions	– *class.stereotype* = «system» ∨ «package» ∨ «process» ∨ «block» – *package.globaldeclaration* = true
Variables	– *class.*<u>*declarationStruct*</u> = *package.*<u>*sdldefinition*</u>

Rule 60. Set Global Declaration Struct

Preconditions	– *class.stereotype* = «system» ∨ «package» ∨ «process» ∨ «block» – *package.globaldeclaration* = false
Variables	– *class.*<u>*declarationStruct*</u> = *class.*<u>*sdldefinition*</u>

Rule 61. Set Local Declaration Struct

Preconditions	– *class.stereotype* = «process» or «block»
Action	Add to *class.*<u>*declarationStruct*</u> : `SIGNALLIST <class.name> = ;`
Variables	– *class.*<u>*signallist*</u> = the added signal list

Rule 62. Create Signal List

Rule 63 translates the generalization relationship between «process» and «block» classes. Because of the preprocessing, super- and subclasses always have the same stereotype, so this is not checked again. The generalization concepts maps very well on the *inherits* concept in SDL, with one exception. In UML, a subclass inherites the association relationship from its superclass, while in SDL a channels and signal routes can not be connected to block or process types and therefore

cannot be inherited. This is solved during the preprocessing by generating extra associations, see Rule 22 and Rule 23.

Preconditions	– *class.stereotype* = «process» ∨ «block» – *class.superclass* ≠ empty – *class.typed* = true
Context	– *structure* = the added block type or process type
Action	Add to signature of *structure*: `INHERITS <class.superclass.name> ;`
Variables	*class.sdlsuper* = *class.superclass.sdldefinition*

Rule 63. Translate Inheritance

Rule 64 translates passive classes into abstract data types. Passive classes are recognized by the stereotype «newtype», as of the SDL keyword for describing data types. This rule only translates the class itself; Rule 94 and Rule 107 respectively translate new attributes new and operations. SDL newtypes do not support inheritance, therefore inheritance between two «newtype» classes is translated in Rule 65 as an attribute "father" in the newtype generated from the subclass.

Preconditions	– *class.stereotype* = «newtype»
Context	– *definedIn* = *class.definedin.sdldefinition*

Action	Add to *class.*:
	`NEWTYPE <class.name> STRUCT`
	`  <here comes the attributes>`
	`  operators`
	`  <here comes the operations>`
	`ENDNEWTYPE <class.name>;`

Variables	– *class.datatype* = the added new type – *signature* = placeholder for translating the attributes – *operator* = placeholder for translating the operators

Rule 64. Translate «newtype» class

Preconditions	– *class.stereotype* = «newtype» – *class.superclass* ≠ empty – *class.superclass.stereotype* = «newtype»
Action	Add to *signature*: `father <class.superclass.name> ;`

Rule 65. Translate Inheritance for «newtype» Classes

V 6.9 Delete Class

Deleting a class in a UML model can have significant consequences on the linked SDL specification. Most changes are indirect however and are translated as such. The attributes and operations in the old (deleted) class, are handled in sections V 6.15 and V 6.18. Association relationships with the deleted class are deleted too, see section V 6.22. The aggregation relationships are deleted in the same way and as a result, the components of the old class are automatically moved to another parent. Finally, subclasses of the old class will loose their inheritance by comparing the old subclass and the new subclass in Rule 75. Rule 66 therefore only has to delete the SDL entities it is directly linked with.

Context for this section	– $class_{old}$ is the old UML class being deleted
Action	Remove $class_{old}.\underline{sdldefinition}$ from its parent Remove $class_{old}.\underline{sdlsignallist}$ Remove $class_{old}.\underline{sdldatatype}$

Rule 66. Translate Deleted Class

After the delete operation, $class_{old}$ keeps a pointer to the original SDL entities. The definition, signallist and datatype are merely deleted from the children and declaration list of the structure containing the information. It is important that $class_{old}.\underline{sdldefinition}$ is not deleted completely because some components of it may still be needed further on, e.g. to move them to another structure.

V 6.10 Compare Class

Class comparison is the most complex compare operation because there are so many things on a class that can change. Translating the changes rely a lot on the semantics regarding the mapping. A typical example is a class changing its stereotype from «block» to «process». This is a drastic change, but a lot of information can be preserved. The process linked to the original class becomes the main definition for the class. Because of the complexity of the class comparison, we divided the translation rules into the different facets.

Rule 67 defines the context variables of the classes to be compared and some extra variables to ease the specification of the translation rules. As with any comparison, the SDL links of the new class are initialized with the links of the old class. In this way, the new class already has the necessary links.

Context for this section	– $class_{old}$ is the old UML class – $class_{new}$ is the new UML class, being compared with $class_{old}$ – $package_{old}$ is the UML package containg $class_{new}$ – $package_{new}$ is the UML package containg $class_{old}$ – $sysclass_{old} = package_{old}.systemclass$ – $sysclass_{new} = package_{new}.systemclass$
Variables	– $class_{new}.\underline{sdldefinition} = class_{old}.\underline{sdldefinition}$ – $class_{new}.\underline{sdlsuper} = class_{old}.\underline{sdlsuper}$

	– $class_{new}.\underline{processesblock} = class_{old}.\underline{processesblock}$ – $class_{new}.\underline{sdlprocess} = class_{old}.\underline{sdlprocess}$ – $class_{new}.\underline{sdlsignallist} = class_{old}.\underline{sdlsignallist}$ – $class_{new}.\underline{sdldatatype} = class_{old}.\underline{sdldatatype}$

Rule 67. Context for Comparing Classes

V.6.10.1 Stereotype has changed

The stereotype of a class actually determines the semantics of the class. Therefore, changing the stereotype can have far-reaching consequences. For example, take a «block» class with some «process» components, which's stereotype is changed into «actor». The linked SDL block is removed and all channels to the block are rewired to the environment. Moreover, the UML aggregations become invalid, the «process» components become top-level and consequently become blocks. Fortunately, most of side affects of the stereotype change are already dealt with during the preprocessing. In the given example, the deleted aggregations, changed components and associations are all translated in their respective translation rules. For this reason, the translation rules can be kept quite simple. The only case where we can really preserve information is switching between the «block» and the «process» stereotype.

Rule 68 defines the translation for a class that switches its stereotype from «block» to «process». The most common case is that the class has already a link to a process. In that case, the process takes over the role of the block. All declarations (signals, sorts, etc.) are moved into the process and the block is replaced by the process. It is possible that after this operation, a block and a process definition appear in the same scope level. This is not allowed in SDL, but this is solved during the SDL post processing explained in section. The other direction, a class changing from «process» to «block» is somewhat easier, see Rule 69.

Preconditions	– $class_{old}.stereotype$ = «block» – $class_{old}.\underline{sdlprocess}$ is not empty – $class_{new}.stereotype$ = «process»
Context	– let $\underline{parent}$ be the structure that contains $class_{new}.\underline{sdldefinition}$
Action	– move declarations of $class_{new}.\underline{sdldefinition}$ into $class_{new}.\underline{sdlprocess}$ – move the complete contents of $class_{new}.\underline{sdldefinition}$ to $\underline{parent}$ – delete $class_{new}.\underline{sdldefinition}$ from $\underline{parent}$
Variables	– $class_{new}.\underline{sdldefinition} = class_{new}.\underline{sdlprocess}$

Rule 68. Change Stereotype «block» to «process»

Preconditions	– $class_{old}.stereotype$ = «process» – $class_{new}.stereotype$ = «block»
Action	– translate $class_{new}$ as a new class – copy the complete contents of $class_{old}.\underline{sdlprocess}$ into $class_{new}.\underline{sdlprocess}$

Rule 69. Change Stereotype «process» to «block»

Preconditions	– $class_{old}.stereotype \neq class_{new}.stereotype$ – Neither Rule 68 or Rule 69 have been fired

Action	– translate $class_{old}$ as a deleted class – translate $class_{new}$ as a new class – stop comparing this class

Rule 70. Stereotype Category Change

V.6.10.2 Typed has changed

The *typed* value of a «block» («process») class determines whether it maps on a block type or a block definition. In other words, if the *typed* value changes to false, the block type is changed into a block definition and the other way around. It is more complicated than that, however. The aggregation to the class has a different semantics for typed and non-typed classes. For a typed class, the aggregation maps on a typed based instance. For a non-typed class, the aggregation only locates to the scope of the definition. Rule 71 and Rule 72 resolve this difference when the *typed* value is changed.

Preconditions	– $class_{new}.stereotype$ = «block» ∨ «process» – $class_{old}.typed$ = false – $class_{new}.typed$ = true
Context	– *structure* = $class_{new}$.*sdldefinition* – *definedin* = *structure.parent*
Action	Transform the block/process $class_{new}$.*sdldefinition* into a block type/process type and move it to $class_{new}$.*definedin*.*sdldefinition*. Add gates G_1, …, G_n to $class_{new}$.*sdldefinition* for each channel/signal route in *structure.parent* going to *structure*. Reuse the name and signal lists of the channel/signal route. In *structure.parent* create a type based block/process instance (*instance*) with name *structure.name* prefixed with "a_" and type *structure.name* Reconnect the channels/signal routes to *instance* by using the newly generated gates

Rule 71. Class becomes Typed

In Rule 72, we look for an instance of the type that is about to be modified in a definition. In normal cases, there is not more than one instance of the class, because the UML preprocessor would mark the class as typed after all. However, if the models are not well synchronized, it may happen that some instances are overlooked. Therefore, we delete the instances in excess. In a tool environment, the user will be queried to agree with the proposed changes.

Preconditions	– $class_{new}.stereotype$ = «block» ∨ «process» – $class_{old}.typed$ = true – $class_{new}.typed$ = false
Context	– let *instance* be an instance of $class_{new}$.*sdldefinition*

Action	Delete all instances of $class_{new}.\underline{sdldefinition}$. Transform the block/process type $class_{new}.\underline{sdldefinition}$ into a block/process definition. If $\underline{instance}$ is not empty, replace $\underline{instance}$ with $class_{new}.\underline{sdldefinition}$ and reconnect the communication and delete other instance of $class_{new}.\underline{sdldefinition}$, else, move $class_{new}.\underline{sdldefinition}$ to $class_{new}.definedin.\underline{sdldefinition}$.

Rule 72. Class becomes Non-Typed

V.6.10.3 Name has changed

To translating a name-change of a class, all the linked SDL entities are renamed. The rules below only have to be executed if the old and the new class have the same stereotype, i.e. both are either «block» or «process». If the stereotype is different, the name change is already handled in the previous section.

Preconditions	– $class_{old}.stereotype$ = «block» ∨ «process» – *and* $class_{new}.stereotype$ = «block» ∨ «process», – *or* $class_{old}.stereotype = class_{new}.stereotype$ = «actor» – $class_{old}.classname \neq class_{new}.classname$
Action	Rename the following entities if not empty: – $class_{new}.\underline{sdldefinition}.name = class_{new}.classname$ – $class_{new}.\underline{sdlprocess}.name = class_{new}.classname$ – $class_{new}.\underline{processesblock}.name$ = <$class_{new}.classname$>`_Processes` – $class_{new}.\underline{signallist}.name = class_{new}.classname$ Update all references to $class_{new}.\underline{signallist}$ (on channels, gates, etc.).

Rule 73. Rename «block» or «process» Class

Preconditions	– $class_{old}.stereotype = class_{new}.stereotype$ = «newtype» – $class_{old}.classname \neq class_{new}.classname$
Action	Rename the following entity: – $class_{new}.\underline{datatype}.name = class_{new}.classname$

Rule 74. Rename «newtype» Class

V.6.10.4 Super Class has changed

In the two rules below we only have to take care about the direct mapping of the inheritance, i.e. the “inherits” clause for types and the “father” entry for newtypes. Side effects such as new or

deleted associations and aggregations are handeled by the preprocessing and the rules about associations and aggregation.

Preconditions	– $class_{new}.stereotype$ = «block» ∨ «process» – $class_{old}.superclass \neq class_{new}.superclass$
Action	If $class_{old}.superclass \neq$ empty, delete the `INHERITS` reference from the signature of $class_{new}.\underline{sdldefinition}$. If $class_{new}.superclass \neq$ empty, add the signature to $class_{new}.\underline{sdldefinition}$: `INHERITS <class.superclass.name> ;`

Rule 75. Change Super Class

Preconditions	– $class_{new}.stereotype$ = «newtype» – $class_{old}.superclass \neq class_{new}.typed.superclass$
Action	If $class_{old}.superclass \neq$ empty, delete the attribute named “father” in $class_{old}.\underline{sdldatatype}$. If such an entry does not exists, delete the attribute of sort $class_{old}.superclass.name$ If $class_{new}.superclass \neq$ empty, add to $class.\underline{datatype}$: `father <class.superclass.name> ;`

Rule 76. Change Super Class for Newtypes

V.6.10.5 Defined in has changed

The *defined in* value of a class defines the scope in which the linked type or definition is located. A different defined in value therefore means that the type must be moved to a different location. The *defined in* value already takes the *global type* option into account, i.e. if *global type* is true, the *defined in* variable is set to the system class for all typed classes.

Preconditions	– $class_{new}.definedin \neq class_{old}.definedin$
Action	If $class_{new}.stereotype$ = «process» and $class_{new}.typed$ = false, move $class_{old}.\underline{sdldefinition}$ into $class_{new}.definedin.\underline{processesBlock}$ Else, move $class_{old}.\underline{sdldefinition}$ into $class_{new}.definedin.\underline{sdldefinition}$

Rule 77. Change Defined In Class

V 6.11 New Aggregation

Adding an aggregation can have more implications than the aggregation itself. For example, adding a second aggregate to a non-type class automatically turns it into a typed class. Again, in the rules below we only have to take care of the direct translation and not of the side effects. Note also that

if an aggregation modeled by the user replaces an aggregation previously added by the preprocessor, it will not be regarded as a new aggregation.

Context for this section.	– *aggregation* is the new UML aggregation – *aggr* is the aggregate class – *comp* is the component class – – *package* is the surrounding package of *class* – *sysclass = package.systemclass* – *system = package.sdlsystem*

Rule 78. Context for Translating Aggregations

If the component class of the new aggregation is typed, a new type based process or block (also called block or process instance) is created, as defined in Rule 79 and Rule 80. Rule 81 translates the case where a «block» or «process» class gets a new «newtype» component. In this case, a variable is added to the process linked with the aggregate class. Similarly, Rule 82 adds an entry to the signature of the aggregate new type. Aggregations between other type of classes, e.g. actor, are considered analysis only and are deleted during preprocessing.

Preconditions	– *comp.stereotype* = «block» – *comp.typed = true*
Context	– *structure = aggr.sdldefinition*
Action	Add to *structure*: `BLOCK a_<comp.name> : <comp.name>;`
Variable	– *aggregation.sdlcomponent* = created type based block

Rule 79. Translate Aggregation with «block» Component

Preconditions	– *comp.stereotype* = «process» – *comp.typed = true*
Context	– *structure = aggr.sdldefinition*
Action	Add to *structure*: `PROCESS a_<comp.name> : <comp.name>;`
Variable	– *aggregation.sdlcomponent* = created type based process

Rule 80. Translate Aggregation with «process» Component

Preconditions	– *aggr.stereotype* = «process» ∨ «block» – *comp.stereotype* = «newtype»
Action	If *aggr. sdlprocess* is not empty, add to *aggr. sdlprocess*: `DCL <aggregation.aggregationrole> <comp.name>;`
Variable	– *aggregation.sdldeclaration* = created variable declaration

Rule 81. Translate Aggregation with New Type Component

Preconditions	– *aggr.stereotype* = « newtype» – *comp.stereotype* = «newtype»
Action	Add to *class.datatype.signature*: `<attribute.name> <attribute.type>;`

Rule 82. Translate Aggregation between New Types

V 6.12 Delete Aggregation

Rule 84 only deletes the SDL component linked to the aggregation if it is a type-based instance, i.e. if the *typed* variable of the component is true. In the case of a process or block definition, deleting the aggregation will cause the definition to move to system level (see Rule 77), not to delete it.

Context for this section.	– $aggregation_{old}$ is the old UML aggregation being deleted – $aggr_{old}$ is the old aggregate class – $comp_{old}$ is the old component class

Rule 83. Context for Deleted Aggregations

Preconditions	– $comp_{old}$.*stereotype* = «block» ∨ «process» – $comp_{old}$.*typed = true*
Context	– *structure* = $aggr_{old}$.*sdldefinition*
Action	Delete *structure* from its parent.

Rule 84. Translate Deleted Instance Aggregation

Preconditions	– *aggr.stereotype* = «process» ∨ «block» – *comp.stereotype* = «newtype»
Context	– *variable* = *aggr*. *sdldeclaration*
Action	Delete *variable* from *aggr.sdlprocess*

Rule 85. Deleted Aggregation with New Type Component

V 6.13 Compare Aggregation

As with all compare rules, in Rule 86 we start by copying the link variables. Rule 87 handles the cases where no information can be reused and the old aggregation is deleted and the new aggregation is added. Rule 88 and Rule 89 respectively update the name and the type of a block or process instance based on the role name and component name. Rule 90and Rule 91 translate the same changes for «newtype» components. We do not need a rule for name changes of the aggregation itself, because the name of the aggregate itself is not used in the translation.

Context for this section.	– $aggregation_{old}$ is the previous UML aggregation – $aggregation_{new}$ is the new UML aggregation to be compared – $aggr_{old} = aggregation_{old}.aggregate$ – $comp_{old} = aggregation_{old}.component$ – $aggr_{new} = aggregation_{new}.aggregate$ – $comp_{new} = aggregation_{new}.component$ – $aggregation_{new}.\underline{sdlcomponent} = aggregation_{old}.\underline{sdlcomponent}$ – $aggregation_{new}.\underline{sdldeclaration} = aggregation_{old}.\underline{sdlcomponent}$

Rule 86. Context for Comparing Aggregations

Preconditions	– $comp_{old} \neq comp_{new}$, or – $aggr_{old}.stereotype \neq aggr_{new}.stereotype$, or – $comp_{old}.stereotype \neq comp_{new}.stereotype$
Action	Translate $aggregation_{old}$ as a deleted aggregation. Translate $aggregation_{new}$ as a new aggregation. Stop comparing these aggregations.

Rule 87. Translate Important Aggregation Change

Preconditions	– $aggregation_{old}.componentRole \neq aggregation_{new}.componentRole$ – $comp_{old}.stereotype = comp_{new}.stereotype$ = «process» or «block» – $comp_{new}$.typed = true
Action	Change the instance name of $comp_{new}.\underline{sdlcomponent}$ into $aggregation_{new}.componentRole$

Rule 88. Translate Component Role Change

Preconditions	– $comp_{old}$.name ≠ $comp_{new}$.name – $comp_{old}.stereotype = comp_{new}.stereotype$ = «process» or «block» – $comp_{new}$.typed = true
Action	Change the type of $comp_{new}.\underline{sdlcomponent}$ into $comp_{new}.name$

Rule 89. Translate Component Type Change

Preconditions	– $aggregation_{old}.componentRole \neq aggregation_{new}.componentRole$ – $comp_{old}.stereotype = comp_{new}.stereotype$ = «newtype»
Action	Change the variable name of $aggregation_{new}.\underline{sdldeclaration}$ into $aggregation_{new}.componentRole$

Rule 90. Translate Role Change to New Type

Preconditions	– $comp_{old}$.name ≠ $comp_{new}$.name

	– $comp_{old}.stereotype = comp_{new}.stereotype$ = «newtype»
Action	Change the type of $aggregation_{new}.\underline{sdldeclaration}$ into $comp_{new}.name$

Rule 91. Translate Name Change of New Type

V 6.14 New Attribute

There are two different translations for an attribute, depending on the stereotype of the class it is defined in. If the class represents a structure, the attribute is translated as variable declaration in the process linked with the class, see Rule 92. If the class is a new type definition, the attribute is translated as an entry in the data type, see Rule 94.

Context for this section	– *attribute* is the new UML attribute – *class* is the surrounding *class* of *attribute*

Rule 92. Context for New Attribute

Preconditions	– *class.stereotype* = «process» ∨ «block» ∨ «system» ∨ «package» – *class.*<u>*sdlprocess*</u> ≠ empty

Action	If *attribute.default* = "", add to *class.*<u>*sdlprocess*</u>
	`DCL <atribute.name> <attribute.type>;`
	else add to <u>*structure*</u>:
	`DCL <atribute.name> <attribute.type> := <attribute.default>;`

Variable	– *attribute.*<u>*declaration*</u> = the added declaration

Rule 93. Translate Attribute in Active Class

Preconditions	– *class.stereotype* = «newtype»
Action	Add to *class.*<u>*datatype.signature*</u>: `<attribute.name> <attribute.type>;`

Rule 94. Translate Attribute in New Type

V 6.15 Delete Attribute

Deleting an attribute is simply translated as deleting the SDL entities that were generated from the original attribute.

Context for this section.	– $attribute_{old}$ is the old UML attribute being deleted – $class_{old}$ is the old class containing $attribute_{old}$

Rule 95. Context for Deleting an Attribute

Preconditions	– $class_{old}.stereotype$ = «process» ∨ «block» ∨ «system» ∨ «package»
Action	Delete $attribute_{old}.\underline{declaration}$

Rule 96. Deleted Variable Declaration

In the case of a new type, the UML attribute has not a pointer to the part of the signature that is linked with the attribute. In Rule 97 we therefore search for a entry in the new type declaration with the same name as the old attribute.

Preconditions	– $class_{old}.stereotype$ = «newtype»
Context	– $\underline{newtype} = class_{old}.\underline{declaration}$
Action	Delete the entry in the signature of $\underline{newtype}$ with name $attribute_{old}.name$

Rule 97. Deleted New Type Entry

V 6.16 Compare Attribute

There are three things of an attribute that can change: the name, the type and the default value. Translating any such change could be implemented as a combination of deleting the old attribute and adding the new attribute. It is possible however that, in UML only the type has changed and that the user has already modified the name of the SDL declaration and refers to this new name in the state chart. Taking the delete and add approach would result in references to a non-existing signal. Therefor we choose to handle each part seperatly.

Context for this section.	– $attribute_{old}$ is the previous UML attribute – $attribute_{new}$ is the new UML attribute to be compared – $class_{old}$ = the class containing $attribute_{old}$ – $class_{new}$ = the class containing $attribute_{new}$

Rule 98. Context for Comparing Attributes

For most UML entities, the first translation rule compares the stereotypes of the entities. Comparing attributes is an exception on this, because the stereotype of an attribute is not used in the mapping definition. The first translation rule handles name changes, see Rule 116.

In Rule 99, Rule 100 and Rule 101 we only check the stereotype of the new class. The stereotype of the new class and the old class are not always the same. If a class changes from «block» to «process», the declarations will automatically be moved to the correct place. Note that in the case the stereotype of a class changes from «block» or «process» to «newtype» or the other way around, the attribute in the new and old classes will not be compared at all. The attributes in the new class will be translated as new attributes. Rule 102 and Rule 103 compares the name and type for the attributes in a «newtype» class.

Preconditions	– $class_{new}.stereotype$ = «process» ∨ «block» ∨ «system» ∨ «package» – $attribute_{old}.name \neq attribute_{new}.name$
Context	– $\underline{variable} = attribute_{old}.\underline{declaration}$
Action	$\underline{variable}.name = attribute_{new}.name$

	Rename all references to the *variable* within the $class_{new}$.*managementProcess*

Rule 99. Translate Attribute Name Change

Preconditions	– $class_{new}.stereotype$ = «process» ∨ «block» ∨ «system» ∨ «package» – $attribute_{old}.type \neq attribute_{new}.type$
Context	– *declaration* = $attribute_{old}$.*declaration*
Action	*declaration.type* = $attribute_{new}.type$

Rule 100. Translate Attribute Type Change

Preconditions	– $class_{new}.stereotype$ = «process» ∨ «block» ∨ «system» ∨ «package» – $attribute_{old}.default \neq attribute_{new}.default$
Context	– *declaration* = $attribute_{old}$.*declaration*
Action	*declaration.initialExpression* = $attribute_{new}.default$

Rule 101. Translate Attribute Default Value Change

Preconditions	– $class_{new}.stereotype$ = «newtype» – $attribute_{old}.name \neq attribute_{new}.name$
Context	– *newtype* = $class_{new}$.*declaration* – *entry* = the entry in *newtype* with name $attribute_{old}.name$
Action	Set the name of *entry* = $attribute_{new}.name$ Update all references to *entry* in all processes to which *newtype* is visible.

Rule 102. Translate Attribute Name Change in «newtype» Class

Preconditions	– $class_{new}.stereotype$ = «newtype» – $attribute_{old}.type \neq attribute_{new}.type$
Context	– *newtype* = $class_{new}$.*declaration* – *entry* = the entry in *newtype* with name $attribute_{old}.name$
Action	Set the type of *entry* = $attribute_{new}.type$

Rule 103. Translate Attribute Type Change in «newtype» Class

V 6.17 New Operation

An operation has different semantics depending on its stereotype: «signal», «procedure» or «operator». A «signal» operation means that the class that contains the operation can receive the specified signal. At the same time it also a declaration of the signal and its parameters. A «procedure» operation is translated as a SDL procedure in the process linked with the class. If the operation is public, the procedure is exported and declared remote so it can be called from outside the process. An «operator» operation becomes a function in a newtype definition.

The rules below do not check the stereotype of the class. Operations with stereotype «signal» and «procedure» can only appear in «process», «block», «system» or «package» classes. Operations with stereotype «operator» can only appear in «newtype» classes. Possible violations to this rule will automatically be corrected during the preprocessing.

Context for this section	– *operation* is the new UML operation – *class* is the class containing *operation*

Rule 104. Context for New Operation

Preconditions	– *operation.stereotype* = «signal»
Context	– *structure* = *class.declarationStruct* – (par_1…par_m) = *operation.parameters*

Action	If m = 0, add to *structure*:
	`SIGNAL <operation.name>;`
	else add to *structure*:
	`SIGNAL <operation.name> ( <par1.type> , … , <parm.type>);`
	Add to *class.signallist*
	`, <operation.name>`

Variable	*operation.sdlsignal* = the generated signal

Rule 105. Translate «signal» operation

Preconditions	– *operation.stereotype* = «procedure»
Context	– *structure* = *class.managementProcess* – *declstruct* = *class.declarationStruct* – (par_1…par_m) = *operation.parameters*

Action	Add to *structure* :
	`EXPORTED PROCEDURE <operation.name>;`
	`FPAR IN <par1.name> <par1.type>,`
	`…`
	`IN <parm.name> <parm.type>,`
	`RETURNS <operation.returntype>;`
	Add to *declstruct* :
	`REMOTE PROCEDURE <operation.name>;`
	`FPAR <par1.type>, … , <parm.type>;`
	`RETURNS <operation.returntype>;`

Variable	*operation.sdlprocedure* = the generated procedure

Rule 106. Translate «procedure» operation

Preconditions	– *operation.stereotype* = «operator»
Context	– *newtype* = $class_{new}$.*declaration* – (par_1…par_m) = *operation.parameters*

Action	Add to the behaviour of *newtype*:
	`<operation.name> <par1.type>, …, <parm.type>`
	`  -> <operation.returntype>;`

Rule 107. Translate «operator» operation

V 6.18 Delete Operation

Context for this section.	– $operation_{old}$ is the old UML operation being deleted – $class_{old}$ is the old class containing $operation_{old}$

Rule 108. Context for Deleting an Attribute

Because of the consistent use of signallists instead of individual signals on channels and signal routes, Rule 109 does not have to care about references to the deleted signal. Of course, it is still possible that the deleted signal is used in some processes, but that is not part of the mapping.

Preconditions	– $operation_{old}$ = «signal»
Context	– *signal* = $operation_{old}$.*sdlsignal* – *signallist* = $class_{old}$.*signallist*
Action	Delete *signal* from *signallist* Delete *signal* from $operation_{old}$.*declarationStruct*

Rule 109. Deleted «signal» operation

Preconditions	– $operation_{old}$ = «procedure»
Context	– *procedure* = $operation_{old}$.*sdlprocedure*
Action	Delete *procedure* from $class_{old}$.*sdlprocess*. Delete the remote procedure definition of *procedure* if it exists.

Rule 110. Deleted «procedure» operation

Preconditions	– $operation_{old}$ = «operator»
Context	– *datatype* = $class_{old}$.*datatype*
Action	Delete the operator named $operation_{old}$.*name* from *datatype*.

Rule 111. Deleted «operator» operation

V 6.19 Compare Operation

Three different aspects of an operation can change: the name, the parameters and the return type. For each of the possible stereotypes («signal», «procedure» and «operator»), these changes are translated differently.

Context for this section.	– $operation_{old}$ is the previous UML operation – $operation_{new}$ is the new UML operation to be compared – ($par_{old}1$, … , $par_{old}n$) = $operation_{old}.parameters$ – ($par_{new}1$, … , $par_{new}m$) = $operation_{new}.parameters$ – $class_{old}$ = the class containing $operation_{old}$ – $class_{new}$ = the class containing $operation_{new}$

Rule 112. Context for Comparing Operations

In contrast with attributes where stereotypes are not used, the stereotype is crucial to the semantics of an operation. If the old and the new operations have a different stereotype, nothing of the declarations can be reused, so the old operartion is deleted and the new operation is added, see Rule 113. As an extra feature, however, we do convert signal outputs into procedure calls if the operation has switch from «signal» to «procedure», see Rule 114 and Rule 115.

Preconditions	– $operation_{old}.stereotype \neq operation_{new}.stereotype$
Action	Translate $operation_{old}$ as a deleted operation Translate $operation_{new}$ as a new operation

Rule 113. Translate Operation Stereotype Change

Preconditions	– $operation_{old}.stereotype$ = «signal» – $operation_{new}.stereotype$ = «procedure»
Action	For all classes associated with $class_{old}$, transform all signal outputs of $operation_{old}$ into procedure calls to $operation_{new}$ with the same parameters.

Rule 114. Translate Operation «signal» to «procedure»

Preconditions	– $operation_{old}.stereotype$ = «procedure» – $operation_{new}.stereotype$ = «signal»
Action	For all classes associated with $class_{old}$, transform all procedure calls to $operation_{new}$ into signal outputs of $operation_{old}$ with the same parameters.

Rule 115. Translate Operation «procedure» to «signal»

The rules below consecutively translate the changes for «signal» operations, «procedure» operations and «operator» operations.

Preconditions	– $operation_{old}.stereotype = operation_{new}.stereotype$ = «signal» – $operation_{old}.name \neq operation_{new}.name$
Context	– $\underline{signal} = operation_{new}.\underline{sdlsignal}$
Action	$\underline{signal}.name = operation_{new}.name$ Rename all references to $\underline{signal}$ (e.g. signal lists, inputs, outputs, saves, etc.) in the complete scope of the signal declaration.

Rule 116. Translate «signal» Operation Name Change

Preconditions	– $operation_{old}.stereotype = operation_{new}.stereotype$ = «signal» – $(par_{old}1.type, \ldots, par_{old}n.type) \neq (par_{new}1.type, \ldots, par_{new}m.type)$
Action	Replace $operation_{new}.\underline{sdlsignal}$ with the following declaration: `SIGNAL <operation.name> ( <par_new1.type> , … , <par_newm.type>);`

Rule 117. Translate Parameter Type Change for «signal» Operation

Preconditions	– $operation_{old}.stereotype = operation_{new}.stereotype$ = «procedure» – $operation_{old}.name \neq operation_{new}.name$
Context	– $\underline{procedure} = operation_{new}.\underline{sdlprocedure}$
Action	$\underline{procedure}.name = operation_{new}.name$ Rename all references to $\underline{procedure}$ (e.g. procedure calls, signallists, etc.) in the complete system.

Rule 118. Translate «procedure» Operation Name Change

Preconditions	– $operation_{old}.stereotype = operation_{new}.stereotype$ = «procedure» – $operation_{old}.returntype \neq operation_{new}.returntype$
Context	– $\underline{procedure} = operation_{new}.\underline{sdlprocedure}$
Action	Set $\underline{procedure.returns} = operation_{new}.returntype$

Rule 119. Translate « procedure » Operation Type Change

Preconditions	– $operation_{old}.stereotype = class_{new}.stereotype$ = «operator» – $(par_{old}1.type, \ldots, par_{old}n.type) \neq (par_{new}1.type, \ldots, par_{new}m.type)$
Action	Regenerate the parameter list of $operation_{new}.\underline{procedure}$, see Rule 106.

Rule 120. Translate Parameter Change of Operator

Preconditions	– $operation_{old}.stereotype = class_{new}.stereotype$ = «operator» – $operation_{old}.name \neq operation_{new}.name$
Context	– $\underline{newtype} = class_{new}.\underline{sdldatatype}$

	– *operator* = the operator in *newtype* with name $operation_{old}.name$
Action	Set the name of *operator* = $operation_{new}.name$

Rule 121. Translate «newtype» Operation Name Change

Preconditions	– $operation_{old}.stereotype = class_{new}.stereotype$ = «operator» – $operation_{old}.type \neq operation_{new}.type$
Context	– *newtype* = $class_{new}.sdldatatype$ – *operator* = the operator in *newtype* with name $operation_{old}.name$
Action	Set *operator.returns* = $operation_{new}.type$

Rule 122. Translate Operation Name Change

Preconditions	– $operation_{old}.stereotype = class_{new}.stereotype$ = «operator» – $(par_{old}1.type, \ldots, par_{old}n.type) \neq (par_{new}1.type, \ldots, par_{new}m.type)$
Context	– *newtype* = $class_{new}.sdldatatype$ – *operator* = the operator in *newtype* with name $operation_{new}.name$
Action	Regenerate the parameter list of *operator*, see Rule 107.

Rule 123. Translate Parameter Change of Operator

V 6.20 Definitions for Associations

This section gives semi-formal definitions for a number of functions that are needed to translate associations. In UML, classes within a package are all "visible" for each other and consequently can have communication associations between any two classes. The SDL instances generated from those classes, however may be invisible for each other. Therefore we need a rerouting mechanism that reroutes direct associations through a number of signal routes and channels. The basis for this mechanism, the (composite) aggregation tree, is defined below.

The first function, "Corresponding type of a class", returns the block type or process type in the SDL specification with the same name as the class, with a priority for block type. It returns nothing if there is no type named like that. Two block types never have the same name because it is not allowed to have two classes with the same name.

Function	CorrespondingType(*class*)
Preconditions	– *package* is a UML package – *class* is a class of *package*
Defnition	If *package.sdlspecification* contains a block type *type*, where *type.name* = *class.name*, then CorrespondingType (*class*) = *type* else, if *package.sdlspecification* contains a process type *type*, where *type.name* = *class.name*, then

	CorrespondingType (*class*) = *type* else CorrespondingType (*class*) = empty

Rule 124. Function Definition for Corresponding Type

The function to calculate the aggregation paths is very important in the translation of associations. An aggregation path is a list of classes that are each other's aggregate and starts with the system class. For a given class, each aggregation path represents an instance or definition of that class. The function *AggrPaths* calculates all possible aggregation paths for a given class. An example shown in Figure V-12 clarifies the functionality. The subsequent functions filter the result of *AggrPaths* for specific purposes.

Function	AggrPaths(*class*)
Preconditions	– *package* is a UML package – *class* is a class of *package* – *class.stereotype* = «system» ∨ «package» ∨ «block» ∨ «process»
Defnition	AggrPaths(*class*) is the set of all possible lists of classes (a_1, a_2, … a_m) with stereotype «system», «package», «block» or «process», where • a_1 = *package.sysclass* • $\forall i = 2..n, \exists$ *aggregation* ∈ *package* : a_{i-1} = *aggregation.aggregate* ∧ a_i = *aggregation.component* ∧ *aggregation.composite* = true • a_m = *class*
Exceptions	– The aggregation paths of an «actor» class "A" equals {(A)} – External classes are treated exactly like non-external classes. This is because the instance of an external class does not differ from normal ones. Note that external classes are not allowed to have composite aggregation components.

Rule 125. Function Definition for Aggergation Path of Class

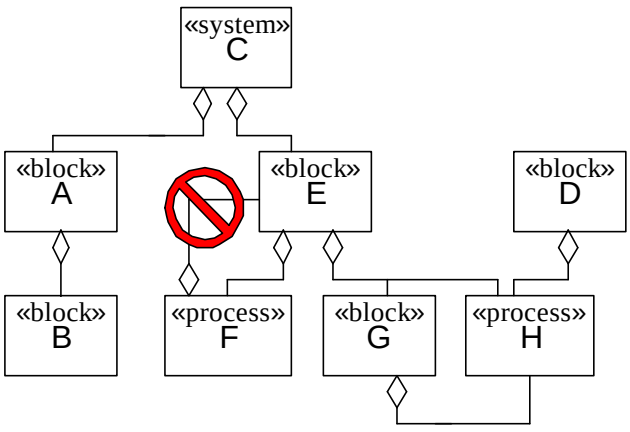

Figure V-12. Aggregation Paths Example

In the example of Figure V-12, the system class has been added explicitly. The «block» class D does not have an aggregate and there will not have an instance in the generated SDL. This is only allowed if class D is types, if not, an aggregation to the system class is automatically added. Note also that aggregation loops are not allowed, so the set of aggregation paths is always finite. In the example we get the following results:

- AggrPaths(C) = {(C)}
- AggrPaths(A) = {(C,A)}
- AggrPaths(F) = {(C,E,F)}
- AggrPaths(H) = {(C,E,H), (C,E,G,H), (D,H)}
- AggrPaths(D) = {(D)}

Function	difference($path_a$, $path_b$)
Preconditions	– *package* is a UML package – $path_a=(a_1,\ldots,a_m)$, $path_b=(b_1,\ldots,b_n)$
Defnition	Let *common* be the maximum for which: $a_{common} = b_{common}$ difference($path_a$, $path_b$) = n + m – 2*common

Rule 126. Function Definition for Difference between Paths

Function	truncate($path_a$, $path_b$)
Preconditions	– *package* is a UML package – $path_a=(a_1,\ldots,a_m)$, $path_b=(b_1,\ldots,b_n)$
Defnition	Let *common* be the maximum for which: $a_{common} = b_{common}$ truncate($path_a$, $path_b$) = (($a_{common},\ldots,a_m$), ($b_{common},\ldots,b_n$))

Rule 127. Function Definition for Truncate Common Paths

Function	CommAggrPaths($class_a$, $class_b$)
Preconditions	– *package* is a UML package – $class_a$ and $class_b$ are classes of *package*
Context	– $paths_a=(PA_1,\ldots, PA_m)$=AggrPaths($class_a$) – $paths_b=(PB_1,\ldots, PB_m)$=AggrPaths($class_b$)
Defnition	– $difference_{i,j}$ = difference(PA_i, PB_j) – Let *min* be the minumum of {i = 1..n, j = 1..n, $difference_{i,j}$} – CommAggrPaths($class_a$, $class_b$) = { truncate(PA_i, PB_j) | i ∈ 1..n, j ∈ 1..m, $difference_{i,j} = min$ }

Rule 128. Conservative Communication Aggregation Path between two classes.

In Rule 128, $PA=(a_1,\ldots,a_m)$ and $PB=(b_1,\ldots,b_n)$ are the two truncated aggregation paths that identify the SDL structures that need to be connected. The result of the function

CommAggrPaths(a,b) is the set of all non-equivalent tuples (A,B) with the least difference, or in other words, closest to each other in terms of scope. An example will clarify this function.

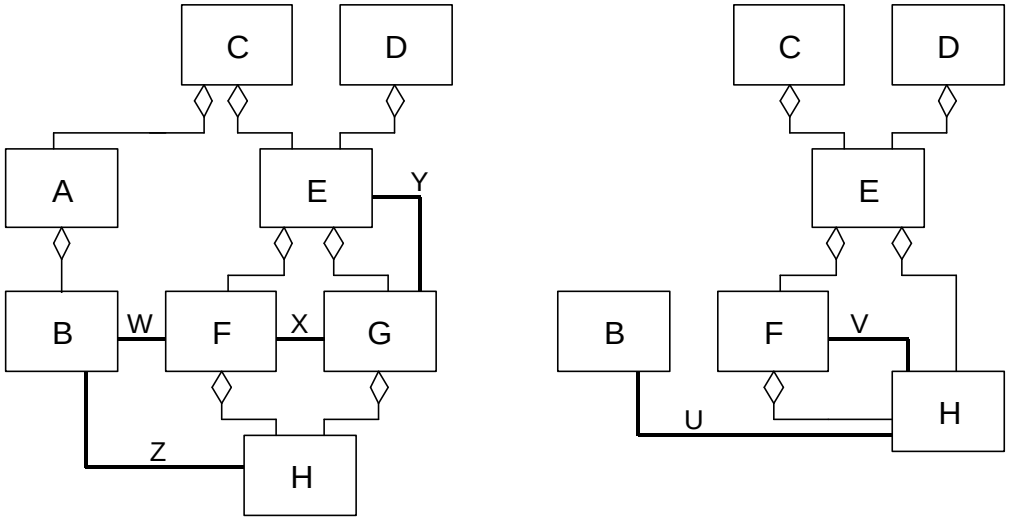

Figure V-13. Illustration to explain Aggregation Paths

In the first example, there are four associations W, X, Y and Z:

- W: CommAggrPaths(B,F) = {((C,A,B), (C,E,F))} and **not** ((C,A,B), (D,E,F))
- X: CommAggrPaths(F,G) = {((E,F), (E,G))}
- Y: CommAggrPaths(E,G) = {((E), (E,G))}
- Z: CommAggrPaths(B,H) = {((C,A,B), (C,E,F,H), ((C,A,B), (C,E,G,H))}

In the second example, there are two associations U and V:

- U: CommAggrPaths(U) = {((B), (C,E,H)), ((B), (D,E,H))}
- V: CommAggrPaths(V) = {((C,E,F), (C,E,F,H)) } and **not** ((C,E,F), (C,E,H)) which has more difference (7-6=1 as opposed to 6-4=2).

The use and interpretation of these aggregation paths are explained in detail later. But basically it is used by the association translation to know which instances should be connected. The function FullCommAggrPaths (Rule 129) is similar to CommAggrPaths, with the only difference that instead of minimizing the difference, it takes all possible combinations. The elimination of equivalent tuples is still done.

Function	FullCommAggrPaths($class_a$, $class_b$)
Preconditions	– *package* is a UML package – $class_a$ and $class_b$ are classes of *package*
Context	– $paths_a$=(PA_1,…, PA_m)=AggrPaths($class_a$) – $paths_b$=(PB_1,…, PB_m)=AggrPaths($class_b$)
Defnition	FullCommAggrPaths($class_a$, $class_b$) = {truncate(PA_i, PB_j) | i ∈ 1..n, j ∈ 1..m}

Rule 129. Full Communication Aggregation Paths between two classes.

The function Instance (Rule 130) returns the corresponding SDL instance given a aggregation path.

Function	*Instance*(path) returns the corresponding SDL instance given a aggregation path.
Preconditions	– *package* is a UML package – *system* = *package.sdlsystem* – *path*=(a_1,…,a_m)
Context	– *aggr* = a_{m-1} – *comp* = a_m
Defnition	If *aggr.typed* = true, let *structure* be the block type or process type in *system* with *aggr.name* as name. Note that within a SDL package there are no two types with the same name. If *aggr.typed* = false, let *structure* be the block or process definition in *system* (or substructure) with *aggr.name* as name. Note that there is only one block or process with this name otherwise aggr would have been typed. Instance(path) = The block or process instance in *structure* with *comp.name* as *name*.

Rule 130. Corresponding SDL Instance.

V 6.21 New Association

The default semantics for an association is communication. In other words, if two classes are connected with an association, it means that some of the instances of one class are able to communicate with some instance of the other class. To achieve the equivalent in SDL, the structures that correspond with the instances are connected with communication routes. For several reasons, translating an associations into channels, signal routes and gates that connect the connect structures is a complicated process. First, the structures to be connected are likely to be in a different scope, so the communication path has to be rerouted through the closest common aggregate. Second, a type based block or process instance can only be connected if the type has the appropriate gates. Moreover, to avoid superfluous gates, they are reused whenever possible.

The translation depends on the communication approach. In the conservative approach, a partial communication route is build at both sides and other communication routes at a higher level are reused if necessary. The goal of this approach is to keep the resulting specification readable and easy to maintain manually. The successive steps to translate a new association for the conservative approach are:

- For each of the two classes connected by the association, find all the combinations of aggregation paths and the corresponding instances.
- Add one gate for each association ends whose class is typed.
- Add two partial channels and/or signal routes, one for each association end.
  - The “from” part of the channels or signal routes is the instance corresponding with the class.

- The “to” part is one of three possibilities: the environment; the instance corresponding with the other association end or the instance that contains the corresponding instance.
- Connect the channels with a gate at both ends if necessary.

In the full connect approach; a complete communication route is build from source to destination. The goal of this approach is to attain code that is complete and can be simulated without further modifications.

- For each of the two classes connected by the association, find all the combinations of aggregation paths and the corresponding instances.
- For each couple of aggregation paths, generate a channel and/or signal route for each part in the non-common path. Use the same “from” & “to” rules as in the conservative approach.

We explain each of these steps in more detail, but we start with the preparation of the signal list declarations for each association and build up the necessary context information.

V.6.21.1 Signallists and Context

All generated gates, channels and signal routes need to declare the signals that can be sent through it. We use the signal lists that are generated per class, see Rule 62. However, the channels generated from an association sometimes need to carry more signal lists than the two classes of the association. In the conservative approach, the “in” signal declarations are the signal lists of all classes in the aggregation tree that has an association with any class in the aggregation tree on opposite association end, including the class itself. The “out” signal declaration is calculated in the opposite direction. Figure V-14 illustrates the situation where both classes A and D have components that communicate with each other. The association between A and D must allow class B to send a signal to class F. The full connect approach is simpler because there is always a full connection between source and destination. In this case, the “in” signal declarations is the signal list of the class. The “out” signal declaration is the signal list of the opposite class.

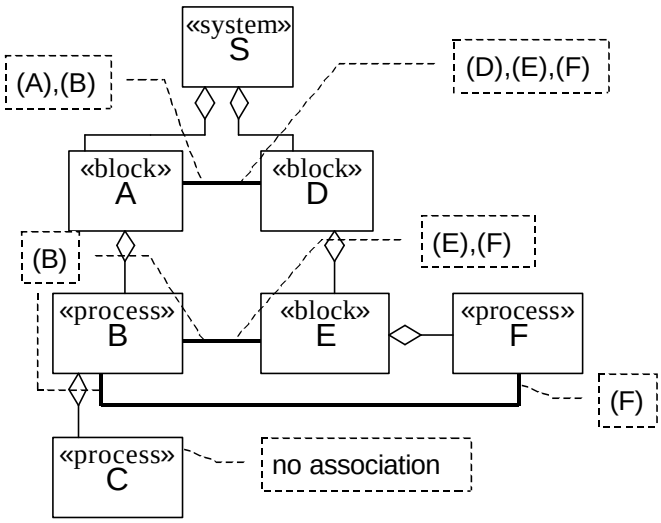

Figure V-14. Illustration for using Class Signal Lists in Conservative Approach

This section, starting with the context in Rule 131, is repeated for every couple of *frompath* and *topath* as a result of *CommAggrPaths* or *FullCommAggrPaths* (for the full communication approach). In Rule 131 and Rule 132, the signal list variables of *association* are calculated. These signal lists do not appear as declarations in the system, but are rather used during the channel and signal route creation.

Preconditions for this section	– *association.stereotype* = «communication»
Context for this section and subsections	– *association* is the new UML association – *fromclass* = *association.fromclass* – *toclass* = *association.toclass* – *frompath* = (fc_1, …, fc_m) – *topath* = (tc_1, …, tc_n) – fc_1= tc_1 (if both are not external), *fromclass* = fc_m, *toclass* = tc_n
Variables	– *association.*<u>*fromsignallist*</u> = (`<fromclass.name>`) – *association.*<u>*tosignallist*</u> = (`<toclass.name>`)

Rule 131. Context for New Associations.

Preconditions	– *communication option* ≠ *full* – *fromclass* is not a deep component of *toclass* – *toclass* is not a deep component of *fromclass*
Context	– CC = {(*class*$_1$, *class*$_2$) | ∃ association between *class*$_1$ and *class*$_2$, *class*$_1$ is a deep component from *fromclass* but not from *toclass*, *class*$_2$ is a deep component from *toclass* but not from *fromclass*}

Action	∀(*class*$_{from}$, *class*$_{to}$)∈CC:
	Add to "*association.*<u>*tosignallist*</u>" :
	`,(<class`$_{to}$`.name>)`
	Add to "*association.*<u>*fromsignallist*</u>" :
	`,(<class`$_{from}$`.name>)`

Rule 132. Add signallists for underlying associations.

Preconditions	– *Fromclass.stereotype* = «process» – *toclass.stereotype* = «process» – *association.variable* = true
Context	– <u>*fromstruct*</u> = *fromclass.*<u>*sdldefinition*</u> – <u>*tostruct*</u> = *toclass.*<u>*sdldefinition*</u>
Action	Add to <u>*fromstruct*</u>: `DCL a_<toclass.name> Pid;`

	Add to *tostruct*: `DCL a_<fromclass.name> Pid;`

Rule 133. Add Pid variable between processes.

V.6.21.2 Generating gates

Before generating any channels or signal routes, we first generate all the gates for each association, with the necessary signal lists on each gate. The basic idea for generating gates is to add a gate to the corresponding type of each association end. In the full-connect scenario, a gate is added to each corresponding type of all the classes on the non-common aggregation path. Figure V-15 illustrates the generated gates with a theoretical example. The association *Z* generates a gate in class *B* and class *F* (i.e. in the structure linked with classes *B* and *F*) because they are both typed and they are the endpoints of the association. In the full connect scenario, class *E* also get a gate

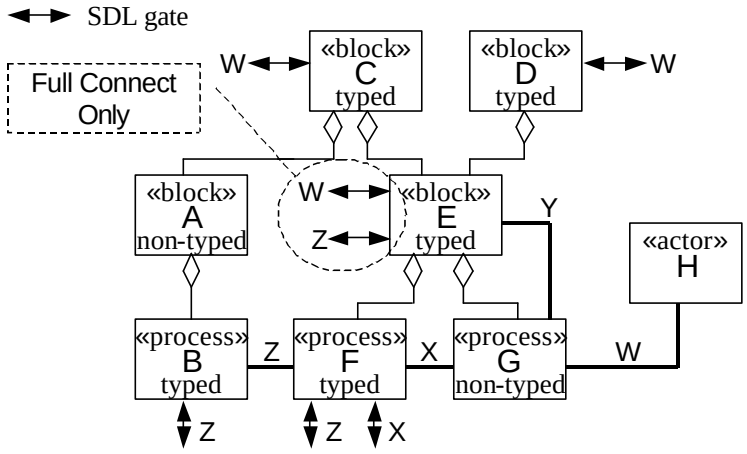

Figure V-15. Example of generated gates

In general, for each association end connected to a typed class, create a gate in the corresponding type. There is an exception in the case that one association end is defined in the scope of the other association end. In this case the former one does not need communication to the outside and thus no gate.

Preconditions	– *communication option = gate only ∨ conservative* – *fromclass.external* = false – *fromclass.typed* = true – *fromclass* ≠ tc_1
Context	– *fromstruct* = *fromclass.sdldefinition*

Action	Add to *fromstruct*:
	`GATE <association.fromrole>`
	`  OUT <association.tosignallist>`
	`  IN <association.fromsignallist>;`
	(skip last line if *association.tofromnavigate*= false)

Rule 134. Add gate in gate-only or conservative communication.

Repeat previous definition for other direction and switch all references to *from* and *to*.

Add a gate to the corresponding type of all classes in the non-common path of the class connected to each association end in respect to the class connected to the other association end.

Preconditions	– *communication option = full*

Action	∀ fc ∈ (fc_2, …, fc_m): if *fc.typed* = true, then add to *fc.sdldefinition*:
	`GATE <association.fromrole>`
	`  OUT <association.tosignallist>`
	`  IN <association.fromsignallist>;`
	(skip last line if *association.tofromnavigate*= false)
	∀ tc ∈ (tc_2, …, tc_m): if *tc.typed* = true, then add to *tc.sdldefinition*:
	`GATE <association.torole>`
	`  OUT <association.fromsignallist>`
	`  IN <association.tosignallist>;`
	(skip last line if *association.tofromnavigate*= false)

Rule 135. Add gates in gate only or conservative communication.

V.6.21.3 Conservative Approach

In the conservative approach, for generating channels and signal routes, the two ends of an associations are treated more or less separately. As shown in Figure V-16, an association to a «process» class A is translated into a signal route starting from a process instance a_A and going into direction of the other end. The other end either can be a process in the same block or can be in an different block. This translation scheme is easier than having to describe all possible combinations block/process, block/actor, system/process, etc.

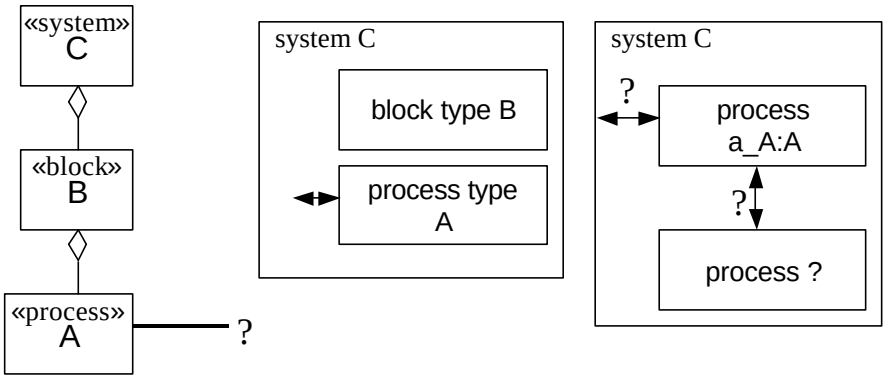

Figure V-16. Illustration of "one-end" translation approach

In the conservative scenario, we generate only one pair of channels and/or signal routes for each communication aggregation path tuple, one for each instance on both ends of the association. If both instances are in the same scope, we generate only one channel or signal route. We start calculating the aggregation paths and the instances. The rules in this section, starting with the context of Rule 136, are executed twice, once for each association end. The second time the *frompath* and *topath* are switched.

Preconditions for this section	– *association.stereotype* = «communication» – *communication option* = conservative

Context for this section	– context from association (*fromclass*, *toclass*, *frompath*, *topath*, *fromsignallist*, *tosignallist*)
	– *frompath* = (fc_1, …, fc_m)
	– *topath* = (tc_1, …, tc_n)
	– $Structure_{from}$ = *Instance*(*frompath*), possibly empty
	– $Structure_{to}$ = *Instance*(*topath*), possibly empty
	
	Determine $Process_{from}$, $Process_{to}$, $Block_{from}$, $Block_{to}$ (default is empty)
	– If *fromclass.stereotype* = «process», then
	$Process_{from}$ = $Structure_{from}$ and
	if (m>1 ∧ fc_{m-1}.*processesblock* ≠ fc_{m-1}.*definitionblock*), then
	$Block_{from}$ = fc_{m-1}.*processesblock* else $Block_{from}$ = empty
	– If *fromclass.stereotype* = «block», then
	$Process_{from}$ = fc_m.*sdlprocess* and
	if (m>1)
	$Block_{from}$ = $Structure_{from}$ else $Block_{from}$ = fc_m.*processesblock*
	– If *toclass.stereotype* = «process»,
	then $Process_{to}$ = $Structure_{to}$,
	else $Process_{to}$ = tc_n.*sdlprocess*
	– Let $Block_{to}$ be an SDL block in the same scope of $Block_{from}$, where $Block_{to}$ is the instance or the definition of one of the classes in *topath*. (May be empty) or is the processes block surrounding $Process_{to}$

Rule 136. Context for Channel and Signal Creation.

For each tuple and for each association end, we have to create some signal routes and/or channels. We will describe the process from the standpoint of class a, if $a_{m-1} \neq b_{n-1}$, the same process should be repeated for the other direction.

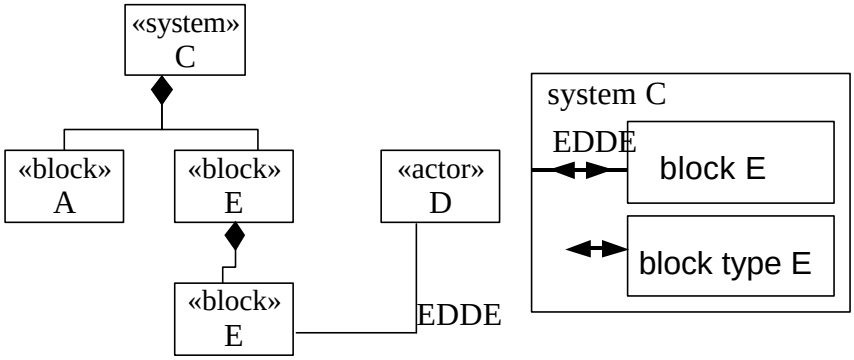

Figure V-17. Communication with External Actor

If the from class is an actor, the aggregation path *frompath* is not important, the instance is considered to be outside the package or system. Therefore a channel is created from the environment of the architecture block into the direction of the destination block or process. Note that in the case of a system, the system itself is actually the architecture block. If the other class is also an actor, no channel is created either.

Preconditions	– $\underline{Process_{from}}$ = empty ∧ $\underline{Process_{to}}$=empty – $\underline{Block_{from}} \neq$ empty ∨ $\underline{Block_{to}} \neq$ empty
Context	– $\underline{struct}$ = $\underline{Instance}$((tc_1,tc_2))

Action	Add to *package*.$\underline{sdlarchitecture}$:
	
	`CHANNEL <association.name>`
	`  FROM ENV TO <struct.name> WITH <tosignallist>;`
	`  FROM <struct.name> TO ENV WITH <fromsignallist>;`
	`ENDCHANNEL <association.name>;`

Rule 137. Add channel for associations from outside

Case 1: n=1. This means that there is no corresponding instance of class a, because $a=a_1$ does not have an aggregate. No instance means no signal route

Case 2: A = B. This means the $Structure_A$ = $Structure_B$ and thus no signal route should be created.

Case 3: $a_{mm-1} = b_{n-1}$, b is «process». This means that classes $Structure_A$ and $Structure_B$ are both processes and are located in the same block. A single signal route is created between $Structure_A$ and $Structure_B$.

Case 4: a_{mm-1} [1] b_{n-1} or b is «block». This means that $Structure_A$ and $Structure_B$ are located in a different block. A signal route is created from $Structure_A$ to the environment. If $Structure_A$ is

located in a “Processes” block, an extra channel should be created from the “processes” block in the direction of *Structure*$_B$: if $a_{mm-1} = b_i \in B$, the channel goes to the corresponding block of b_{i+1}, otherwise the channels goes to the environment.

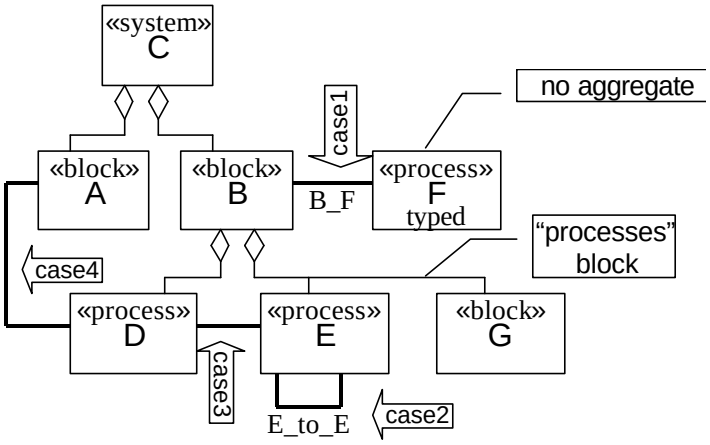

Figure V-18. Illustration of cases for associations to «process» classes

Preconditions	– $\underline{Process_{from}} \neq$ empty $\wedge$ $\underline{Process_{to}} \neq$ empty $\wedge$ $\underline{Process_{from}} \neq \underline{Process_{to}}$ – $\underline{Process_{from}}$ and $\underline{Process_{to}}$ are in the same scope
Context	– $\underline{struct}$ = parent structure of $\underline{Process_{from}}$

Action	Add to $\underline{struct}$:
	
	`SIGNALROUTE <`*`association.name`*`>`
	`  FROM <`$\underline{Process_{from}}$`.name> TO <`$\underline{Process_{to}}$`.name>`
	`  WITH <`$\underline{tosignallist}$`>;`
	`  FROM <`$\underline{Process_{to}}$`.name> TO <`$\underline{Process_{from}}$`.name>`
	`  WITH <`$\underline{fromsignallist}$`>;`

Rule 138. Add signal route between two processes in the same scope.

Preconditions	– $\underline{Process_{from}} \neq$ empty – $\underline{Process_{to}}$ = empty $\vee$ $\underline{Process_{from}}$ and $\underline{Process_{to}}$ are in a different same scope
Context	– $\underline{struct}$ = the parent structure of $\underline{Structure_{from}}$

Action	Add to $\underline{struct}$:
	
	`SIGNALROUTE <`*`association.name`*`>`
	`  FROM <`$\underline{Process_{from}}$`.name> TO ENV`
	`  WITH <`$\underline{tosignallist}$`>;`
	`  FROM ENV TO <`$\underline{Process_{from}}$*`.name`*`>`
	`  WITH <`$\underline{fromsignallist}$`>;`

Rule 139. Add signal route from process to the environment.

Preconditions	– $\underline{Process_{from}} \neq$ empty $\wedge$ $\underline{Block_{from}} \neq$ empty – $\underline{Block_{from}}$ is *not* the parent structure of $\underline{Process_{from}}$
Context	– Let *struct* be the parent structure of $\underline{Process_{from}}$

Action	Add to $\underline{Block_{from}}$:
	`CHANNEL <association.name>`
	`  FROM <struct.name> TO ENV`
	`  WITH <tosignallist>;`
	`  FROM ENV TO <struct.name>`
	`  WITH <fromsignallist>;`
	`ENDCHANNEL;`

Rule 140. Add chanel from processes block to the environment.

Preconditions	– $\underline{Block_{from}} \neq$ empty $\wedge$ $\underline{Block_{to}} \neq$ empty $\wedge$ $\underline{Block_{to}} \neq \underline{Block_{from}}$ – $\underline{Block_{from}}$ and $\underline{Block_{to}}$ are in a different same scope
Context	– Let *struct* be the parent structure of $\underline{Block_{from}}$

Action	Add to *struct*:
	`CHANNEL <association.name>`
	`  FROM <Block_from.name> TO <Block_to.name>`
	`  WITH <tosignallist>;`
	`  FROM <Block_to.name> TO <Block_from.name>`
	`  WITH <fromsignallist>;`
	`ENDCHANNEL;`

Rule 141. Add signal route between two blocks in the same scope.

Preconditions	– $\underline{Block_{from}} \neq$ empty – $\underline{Block_{to}}$ = empty $\vee$ $\underline{Block_{from}}$ and $\underline{Block_{to}}$ are in a different same scope
Context	– Let *struct* be the parent structure of $\underline{Block_{from}}$

Action	Add to *struct*:
	`CHANNEL <association.name>`
	`  FROM <Block_from.name> TO ENV`
	`  WITH <tosignallist>;`
	`  FROM ENV TO <Block_from.name>`
	`  WITH <fromsignallist>;`
	`ENDCHANNEL;`

Rule 142. Add signal route from block to the environment.

Case 1: n=1. This means that there is no corresponding instance of class a, because $a=a_1$ does not have an aggregate. No instance means no signal route

Case 2: A = B. This means the $Structure_A$ = $Structure_B$ and thus no signal route should be created.

Case 3: a_{nm-1} = b_{n-1} Ù b is «block». This means that classes $Structure_A$ and $Structure_B$ are both

blocks and are located in the same structure. A single channel is created between $Structure_A$ and $Structure_B$.

Case 4: a_m Î B. This means that classes $Structure_B$ is defined within $Structure_A$. No channel is created from $Structure_A$. If $Structure_A$ contains a management process, see below.

Case 5: other. This means that $Structure_A$ and $Structure_B$ are not located in the same structure. A channel is created from $Structure_A$ in the direction of $Structure_B$: either to the environment or to a neighbor block containing $Structure_B$.

Management process, for case 3, 4 & 5: If in cases 3, 4 or 5, $Structure_A$ has a "management" process, an extra signal route should be created from the management process in the direction of $Structure_B$. If this management process is in a "processes" block and $Structure_B$ is not defined in the processes block, then an extra channel should be created from the processes block in the direction of $Structure_B$.

V.6.21.4 Full Connect Channel & Signal Route Generation

In the full connect scenario, we generate a full connection from one instance to another, using channels and/or signal routes for each part in the communication aggregation path. If both instances are in the same scope, we generate only one channel or signal route. As in the conservative approach, we start calculating the aggregation paths and the instances.

Preconditions for this subsection	– *association.stereotype* = «communication» – *communication option* = conservative
Context for this section	– same context from association (*fromclass*, *toclass*, *frompath*, *topath*, *fromsignallist*, *tosignallist*) – *frompath* = (fc_1, …, fc_m) – *topath* = (tc_1, …, tc_n) – $Structure_{from}$ = *Instance*(*frompath*), possibly empty – $Structure_{to}$ = *Instance*(*topath*), possibly empty

Rule 143. Context for New Associations.

In Rule 144, we reuse the channel and signal route generation of the conservative approach to connect the process and the processes-block. In many cases, this will already result in a full connection, e.g. for two processes within the same scope. The advantage is that in the other rules (Rule 145 through Rule 147), we only have to take care about the channel generation between the block structures. Rule 145 takes the second classes in both aggregation paths and connects the linked structure with a channel. The aggregate of these classes is the first common aggregate of the instances to be connected. Rule 146 and Rule 147 build channels starting from the structures of Rule 145 down to the structure linked with the initial classes. Figure V-19

Action	Execute Rule 136 through Rule 142 of the conservative approach.

Rule 144. Reuse Conservative Translation

Preconditions	– $fc_1 = tc_1 \wedge m>2 \wedge n>2$
Context	– $\underline{Block_{from}} = \underline{Instance}((fc_1,fc_2))$ – $\underline{Block_{to}} = \underline{Instance}((tc_1,tc_2))$ – $\underline{struct} = fc_1.\underline{sdldefinition}$ (the parent structure of $\underline{Block_{from}}$ and $\underline{Block_{to}}$)

Action	Add to *struct*:
	`CHANNEL <association.name>`
	`  FROM <Block_from.name> TO <Block_to.name>`
	`  WITH <tosignallist>;`
	`  FROM <Block_to.name> TO <Block_from.name>`
	`  WITH <fromsignallist>;`
	`ENDCHANNEL;`

Rule 145. Add signal route between two blocks in the same scope.

Preconditions	– $m > 3$
Context	– $\forall i \in (3..m-1)$: execute this rule, where $\underline{block_{from}} = \underline{Instance}((fc_1,\ldots,fc_i))$ – Let *struct* be the parent structure of $\underline{Block_{from}}$

Action	Add to *struct*:
	`CHANNEL <association.name>`
	`  FROM <Block_from.name> TO ENV`
	`  WITH <tosignallist>;`
	`  FROM ENV TO <Block_from.name>`
	`  WITH <fromsignallist>;`
	`ENDCHANNEL;`

Rule 146. Add signal route from block to the environment.

Preconditions	– $n > 3$
Context	– $\forall i \in (3..n-1)$: execute this rule, where $\underline{block_{to}} = \underline{Instance}((tc_1,\ldots,tc_i))$ – Let *struct* be the parent structure of $\underline{Block_{to}}$

Action	Add to *struct*:
	`CHANNEL <association.name>`
	`  FROM ENV TO <Block_to.name>`
	`  WITH <tosignallist>;`
	`  FROM <Block_to.name> TO ENV`
	`  WITH <fromsignallist>;`
	`ENDCHANNEL;`

Rule 147. Add signal route from block to the environment.

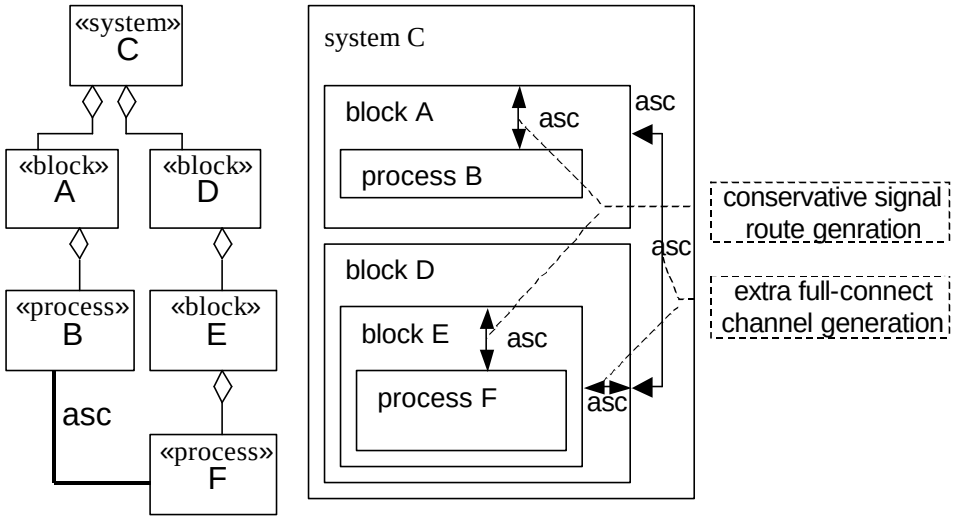

Figure V-19. Example of Channel Generation in Full Connect

V.6.21.5 Connecting Channels and Signal Routes

Because of the complexity and the customization of the translation rules, it is very difficult to predict to which gate or channel a channel or signal route will be connected. In this separate connecting phase, we search for the gate in a type or a channel in the environment that fits best. This search is done based on the in/out signallist, on the original value of the connection and on the name of the channel to be connected.

For each end of a channel or signal route:

– If going to the environment of a type search for a gate
– If going to the environment of a definition, search for a channel
– If going to a type based instance, search for a gate

The following priorities hold:

1. If the default connect given during translation exists (gate or channel), keep that connect.
2. Search for a gate or channel in the environment with the same name
3. Search for a gate or channel with exactly the same in/out signal lists.
4. Search for a gate or channel with contains most of the in/out signal lists, possibly zero.

V 6.22 Delete Association

A deleted association is translated by deleting the gates, channels and signal routes that are linked with the association. After this step, other channels or signal routes that were connected with the linked entities now become unconnected. This is solved by performing the connection process described in section V.6.21.5.

Context for	– $association_{old}$ is the old UML association being deleted

this section.	

Rule 148. Context for Deleting an Association

Action	Delete *association.sdlfromroute* Delete *association.sdltoroute* Delete all channels in *association.sdlfromchannels* Delete all channels in *association.sdltochannels* Delete all gates in *association.sdlgates*

Rule 149. Deleted association

V 6.23 Compare Association

Translating associations is very complex process and apply a small change to the association can have many implications on the generated SDL. For example, if one ends of an association is connected to a different class, possibly none of the previously generated channels can be reused (Rule 152). Such a change is translated as deleting the old association and adding the new association. Other changes, like renaming the association, can be translated gracefully. The translation rules start with the worst cases.

Context for this section.	– $association_{old}$ is the previous UML association – $association_{new}$ is the new UML association to be compared

Rule 150. Context for Comparing Associations

Preconditions	– $association_{old}.stereotype \neq association_{new}.stereotype$
Action	Translate $association_{old}$ as a deleted association Translate $association_{new}$ as a new association

Rule 151. Translate Association Stereotype Change

Preconditions	– $association_{old}.fromclass \neq association_{new}.fromclass$ or – $association_{old}.toclass \neq association_{new}.toclass$
Action	Translate $association_{old}$ as a deleted association Translate $association_{new}$ as a new association

Rule 152. Translate Association Ends Change

Rule 153 translate a name change of the association. Note that this also covers the case where the *from* and/or *to* classes are renamed and where the name of the association is derived from the class names.

Preconditions	– $association_{old}.name \neq association_{new}.name$
Context	– $newname = association_{new}.name$
Action	For all *channel* ∈ *association.sdlchannels* do *channel.name = newname* Set *association.sdlfromroute.name = newname* Set *association.sdltoroute.name = newname*

Rule 153. Translate Association Rename

Preconditions	– $association_{old}.fromrolename \neq association_{new}.fromrolename$
Context	– $newname = association_{new}.fromrolename$
Action	∀ *gate* ∈ *association.sdlgates* | ∃ *channel* ∈ *association.fromchannels* : *channel* is connected with *gate*, do *gate.name = newname*

Rule 154. Translate From-Role Rename

Preconditions	– $association_{old}.torolename \neq association_{new}.torolename$
Context	– $newname = association_{new}.torolename$
Action	∀ *gate* ∈ *association.sdlgates* | ∃ *channel* ∈ *association.tochannels* : *channel* is connected with *gate*, do *gate.name = newname*

Rule 155. Translate To-Role Rename

V 6.24 New State Diagram

A UML class can contain exactly one state diagram. As such, the class and its state diagram can be considered as one entity. This matches with the situation in SDL where the process is a structure as well as a container for states. Accordingly, a UML class and its state diagram both have a link to the same SDL process. For the translation, this means that the state diagram itself does not need any translation, because the process is already managed while translating the UML class. In Rule 156, we therefore only set the *sdlprocess* link variable to the same value as the class.

Context for this section	– *statediagram* is the new state diagram – *class* is the class containing *statediagram*
Action	Set *statediagram.sdlprocess = class.sdlprocess*

Rule 156. Set *sdlprocess* link for State Diagram

V 6.25 Delete State Diagram

Deleting a state diagram does not necessarily mean that the linked process must be deleted. The process can still contain declaration and procedures. As explained in the previous section, it is the responsibility of the class translation to manage the existence and properties of the linked process. However, by deleting the state diagram, all states in it are implicitly deleted too. Because the hierarchical comparison does not compare state if there are no matching state diagrams,

Context for this section	– *statediagram* is the delete diagram – *process* = *statediagram.sdlprocess*
Action	Delete the initial transition and all states from *process*

Rule 157. Set *sdlprocess* link for State Diagram

V 6.26 Compare State Diagram

There are no attributes on a UML state diagram that can be modified, so there are no comparison rules for state diagrams. Of course, the states and transitions contained in the matching state diagrams are compared. The translation rules for these changes are given below.

V 6.27 New State

The basic state and transition concepts are very similar in UML and SDL and thus are easy to translate. Rule 159 translates only normal states. Start states and stop state do not have to be translated separately. The extra features of UML, like nested states and entry/exit actions, are already converted to basic concepts during the preprocessing of the UML model.

Context for this section	– *state* is the new state – *statediagram* is the state diagram containing *state* – *class* is the class containing *statediagram* – *sdlprocess* = *class.sdlprocess*

Rule 158. Context for Translating State

Preconditions	– *state.type* = normal (i.e. not start or stop)

Action	Add to *sdlprocess*:
	`STATE <state.name>;`
	`ENDSTATE <state.name>;`

Variables	*state.sdlstate* = the state created above

Rule 159. Translate new State.

V 6.28 Delete State

Context for	– $state_{old}$ is the deleted state

this section	– $statediagram_{old}$ is the state diagram containing $state_{old}$ – *sdlprocess* = *class.sdlprocess*

Rule 160. Context for Deleting State

Preconditions	– *state.type* = normal (i.e. not start or stop)
Action	Delete $state_{old}$.sdlstate from $statediagram_{old}$.sdlprocess

Rule 161. Translate Deleted State

V 6.29 Compare State

The only property in a UML state that can change is its name. Rule 162 gibves the context and copies the state-link to the new UML state and Rule 163 translates the renaming if applicable.

Context for this section.	– $state_{old}$ is the previous UML state – $state_{new}$ is the new UML state
Variables	– $state_{new}$.*state* = $state_{old}$.*state*

Rule 162. Context for Comparing States

Preconditions	– $state_{old}.name \neq state_{new}.name$
Action	Set $state_{new}$.*state*.*name* = $state_{new}$.*name* Translate $operation_{new}$ as a new operation

Rule 163. Translate State Rename

V 6.30 New Transition

The transition in the UML data structure is built in correspondence with SDL, where a transition is represented by the input and/or guarded symbol. Except for the source and destination state, a UML transition also holds the input event that triggers the transition and/or the guard that gives the condition that must be fulfilled to execute the transition. Moreover, during the preprocessing, the entry and exit actions of the states are copied to the transitions. As a result, the rules below have all the necessary information in the transition itself.

Context for this section	– *transition* is the new transition – *source* = *transition.source* – *dest* = *transition.dest* – *statediagram* is the state diagram containing *source* – *class* is the class containing *statediagram* – *sdlprocess* = *class.sdlprocess*

Rule 164. Translate Context for State Diagram

A transition coming from a start state, must always be a event-less and guard-less event. So there is only one possible translation. In Rule 165, the *sdltransition* link of the UML start transition is filled in so that actions can be added later on. For transitions starting from a normal state, there are three basic types: event, when and after. An *event* transition can also contain a when clause (guarded event). Rule 166 till Rule 168 translates the trigger of the transition and Rule 169 till Rule 171 translates the destination of the transition. Note that the list of actions in the transition (*transition.actions*) are translated individually as new actions.

Preconditions	– *source.type* = start – *sdlprocess* does not contain a start transition
Action	Add to *sdlprocess*: `START;` In terms of the information model: *sdlprocess.start* = new transition
Variables	*transition.sdltransition* = the start transition created above

Rule 165. Translate Start Transition.

Preconditions	– *source.type* = normal – *transition.type* = event
Context	– *sdlsource* = *source.sdlstate*

Action	Add to *sdlsource*:
	`INPUT <tranistion.event>;`
	If *transition.guard* is not empty, add to *sdlsource*:
	`PROVIDED <tranistion.guard>;`

Variables	*transition.sdltransition* = the transition created above

Rule 166. Translate Normal Transition.

Preconditions	– *source.type* = normal – *transition.type* = when
Context	– *sdlsource* = *source.sdlstate*
Action	Add to *sdlsource*: `PROVIDED <tranistion.guard>;`
Variables	*transition.sdltransition* = the transition created above

Rule 167. Translate Guarded Transition.

Preconditions	– *source.type* = normal – *transition.type* = after
Context	– *sdlsource* = *source.sdlstate*

Action	Add to *process* :
	`TIMER <transition.event> := <transition.timer>;`
	Add to *sdlsource* :
	`INPUT <tranistion.event>;`

Variables	*transition.sdltransition* = the transition created above (= second code line)

Rule 168. Translate Timer Transition.

Preconditions	– *transition.isinternal* = true
Action	Add to *transition.sdltransition* : `NEXTSTATE -;`

Rule 169. Next State for Internal Transition.

Preconditions	– *dest.type* = normal – *transition.isinternal* = false
Action	Add to *transition.sdltransition* : `NEXTSTATE <dest.name>;`

Rule 170. Normal Next State.

Preconditions	– *dest.type* = stop – *transition.isinternal* = false
Action	Add to *transition.sdltransition* : `STOP;`

Rule 171. Transition to Stop State

V 6.31 Delete Transition

Context	– $transition_{old}$ is the deleted transition – $state_{old}$ is the state containing $transition_{old}$ – *sdlprocess* = *class.sdlprocess*
Action	Delete $transition_{old}$.*transition* from $state_{old}$.*state*

Rule 172. Translate Deleted Transition

V 6.32 Compare Transition

Many properties of a transition can be changed. Some changes are very hard to translate and require some overwriting, such as an internal transition that changes into an external transition (Rule 174) and a transition that changes the type of the trigger (Rule 175). A change of the source and/or destination state is easier to translate (Rule 176 and Rule 177).

Context for this section.	– $transition_{old}$ is the previous UML transition – $transition_{new}$ is the new UML transition
Variables	– $transition_{new}.\underline{transition} = transition_{old}.\underline{transition}$

Rule 173. Context for Comparing Transitions

Preconditions	– $transition_{old}.isinternal \neq transition_{new}.isinternal$ **or** – $transition_{old}.source.type \neq transition_{new}.source.type$ (from start to normal)
Action	Translate $transition_{old}$ as deleted and $transition_{new}$ as new

Rule 174. Translate Major Transition Change

Preconditions	– $transition_{old}.type \neq transition_{new}.type$
Action	Regenerate the input and the guard symbols from $transition_{old}.\underline{transition}$ by using Rule 166 until Rule 168.

Rule 175. Translate Transition Type Change

Preconditions	– $transition_{old}.source \neq transition_{new}.source$
Action	Move $transition_{old}.\underline{transition}$ to $transition_{new}.source.\underline{state}$

Rule 176. Translate Transition Move

Preconditions	– $transition_{old}.dest \neq transition_{new}.dest$
Action	Delete the **nextstate** statement from $transition_{new}.\underline{transition}$ and reapply Rule 170 or Rule 171.

Rule 177. Translate Transition Destination Change

V 6.33 New Action

Only new actions are translated, not deleted actions. Moreover, actions are not compared and as a result, modifying an action will result in a new action. There are two main reasons for this. First, actions do not have a unique identification; they are merely sub-strings in the action string. Consequently, it is impossible to determine whether an action has been modified or an action was deleted and added. Second, it is typically not the intention in a UML state diagram to write full SDL code on the transitions, but rather a more informal description. During detailed design the

generated SDL actions will therefore be modified a lot, making it difficult to maintain the link and to translate changes in a sensible way.

Rule 179 through Rule 182 determine the type of the action by checking for a certain sub-string. If no special string is found, the default translation is an SDL task with free text.

Context for this section	– *action* is the new action – *transition* is the action containing *action*

Rule 178. Translate Context for Action

Preconditions	– *action.name* starts with "^"
Variables	– Parse *action.name* as *^destination.signal* or *^signal* ; possible parameters within parentheses are included in *signal*

Action	If destination is empty, add to *transition.sdltransition* :
	`OUTPUT <signal>;`
	else,
	`OUTPUT <signal> TO <destination>;`

Rule 179. Translate Signal Send Action.

Preconditions	– *action.name* starts with "call"
Action	Add to *transition.sdltransition* : `<action.name>;`

Rule 180. Translate Call Action.

Preconditions	– *action.name* includes the string ":="
Action	Add to *transition.sdltransition* : `TASK <action.name>;`

Rule 181. Translate Assignment Action.

Preconditions	– Rule 179, Rule 180 and Rule 181 do not apply.
Action	Add to *transition.sdltransition* : `TASK '<action.name>';`

Rule 182. Translate Free Text Action.

V. 7 SDL post processing

After applying the translated changes, we check the resulting SDL specifications for inconsistencies or violations of the SDL syntax that are typical introduced by the translation. This "cleanup" is not part of the translation rules of the previous section for two reasons. First, we do not want to overload the translation rules with recurring aspects. For example, every rule adding or deleting a block or process should check whether there is not a block or process next to each other. Second, a problem introduced by one translation rule may already be solved be the application of a subsequent rule.

V 7.1 Structure

The main structural post-processing is to eliminate processes and blocks in the same scope. According to the SDL rules, a block and a process can never be in the same scope. Usually, this is avoided by creating an extra "processes block". During iteration, however, it is for example possible that a block was added to a process-only block. Rule 183 therefore checks the coexistence of blocks and process in all blocks and solves it accordingly. Rule 184 performs the opposite action, i.e. delete the processes block if it is no longer necessary. Rule 183 and Rule 184 must be checked for all *block* definitions and block types in the complete specification.

Preconditions	– *block* contains at least one block definition or block intance – *block* contains at least one process definition or process instance
Context	– *class* = *block.sdldefinition*$^{-1}$
Action	Execute Rule 55 to add a processes-block *processes* in *block*. Move all process and signal routes of *block* into *processes*: For all moved signal routes that go to environment, create an equivalent channel in *block*.
Variables	– *class.processesblock* = *processes*

Rule 183. Eliminate Blocks and Processes in the Same Scope.

Preconditions	– *block.processesblock*$^{-1}$ = *block.parent.sdldefinition*$^{-1}$ – Except for *block* there are no other blocks, block instances, processes or process instances in its scope – *block.parent* ¹ *block.system*
Context	– *class* = *block.sdldefinition*$^{-1}$
Action	Delete all channels from *block.parent*. Move the complete contents of *block* into *block.parent*. Delete *block* from *block.parent*.
Variables	– *class.processesblock* = *class.sdldefinition*

Rule 184. Eliminate Unnecessary Processes-Blocks.

V 7.2 Communication

During the UML post processing, the channels and signal routes are connected with the generated gates. In a one shot translation, the gates are generated first and the channels are immediately connected to the best gate. Because gates can disappear during iteration, we connect channels and signal routes with gates during post processing.

Context for this section	– *communication* is a channel or signal route – *parent* is the structure that encloses *communication*

Rule 185. Context for Connecting Commuincations

Preconditions	– *communication.fromconnect* is not a valid gate – *communication.fromstruct* = “ENV” – *parent* is a block type
Action	Set *communication.fromconnect* to the gate of *parent.gates* that best matches the input and output signals of *communication*.

Rule 186. Connect Communication with Environment Gate.

Preconditions	– *communication.fromconnect* is not a valid gate – *communication.fromstruct* is a block instance or process instance
Context	– *structtype* is the block or process type of *communication.fromstruct*
Action	Set *communication.fromconnect* to the gate of *structtype.gates* that best matches the input and output signals of *communication*.

Rule 187. Connect Communication with Best Gate.

V 7.3 Declarations

There are two more issues concerning declarations. First, if a declaration is defined twice, the one in the highest scope is retained, see Rule 188. Rule 189 adds a default signal to signal lists that do not contain any signals yet. This is necessary for active classes that do not contain operations.

Preconditions	– *decl* is a signal, signal list or newtype declaration – The same declarations is already defined in a higher scope
Action	Delete *decl*.

Rule 188. Connect Communication with Best Gate.

Preconditions	– *signallist* is a signal list without any signals defined in it.
Action	Add the signal “empty” to *signallist*. If the “empty” signallist is not defined yet, add a signal with name “empty” and no parameters to the system.

Rule 189. Add default signal to empty signal lists.

V. 8 SDL to UML

V 8.1 Reverse Iteration

This section defines the rules to translate changes in SDL into changes in UML. The intention is to update the system design document with the important changes made during detailed design, whether that be added, modified or deleted information. The approach is very similar to the forward iteration. The old and the new version of an SDL specification are compared in a structural way, i.e. compare data structures instead of ASCII texts. Changes

Unlike the forward iteration, the reverse iteration is not intended for the translation of a complete specification. We assume that there has been carried out a forward translation before and that the UML model and SDL specification are synchronized before the reverse iteration starts. This is a reasonable assumption for projects where UML and SDL are used right from the start. For legacy systems for which only the SDL specification exists, a complete reverse engineering must be performed first. Most of the "new" translation rules presented below can be reused for that purpose, but they need to be extended with rules to generate diagrams, create packages and their relationships, find communication patterns, abstract or summarize state charts and link blocks with a process. Furthermore, these extra rules need customization options to be adaptable to the goal of the reverse engineering. We decided that a standalone reverse engineering is out of the scope of this dissertation.

V 8.2 UML Model versus Diagrams

Unlike SDL, UML has no strict correspondence between the class diagrams and the information contained in the model. In SDL, the only difference between the graphical notation and the textual notation is that the graphical notation uses extra positioning information. If this positioning information lacks, an SDL tool will perform an automatic layout. In UML, a class, an association or an inheritance relationship may be shown in several diagrams or may not be shown at all. In other words, diagrams are optional and usually show only a part of the information in the model. For this reason, the UML-SDL mapping rules are defined on the UML information model and not on the diagrams. Section V 9.1 illustrates the generation of some diagrams based on the translated changes.

V 8.3 Specification & Packages

As argued in section V 8.1, we do not support full reverse engineering. In a round-trip scenario, a UML model with packages is created first and translated to SDL before starting to iterate. Similarly, new packages should be added on UML level in order to include them in the round-trip process. Of course, developers are free to import other SDL packages in their system; these packages will simply not be translated to UML as a change. Rule 190 therefore only defines a number of context variables used throughout the SDL to UML translation.

Context for	– $\underline{system_{old}}$ is the previous version of the SDL system or package

this section	– *system*$_{new}$ is the new and modified version of the SDL system or package – *system* = *system*$_{new}$ as a shorthand – *package* is the UML package that needs to be updated – *sysclass* = *package.systemclass* – *architecture* = *package.sdlarchitecture* is the structure that contains the instances of top-level classes.

Rule 190. Translate Context for Reverse Iteration

V 8.4 New Block

The standard translation for a new block is to create a new class with stereotype «block» in the UML package. For a new block instance, only an aggregation to the class that is linked with the block type is created. However, for block instances of an imported block type this class is not available yet. In this case, the block instance is translated as a new external class, see Rule 192. Another exception is the processesblock (a block which only purpose is to hold te management process), which is not translated at all.

The translation of the specialization construct into a generalization relationship is done separately in section V 8.10.

Context for this section	– *block* is the new SDL block, block type or block instance – *parent* is the surrounding SDL structure of *block* – *parentclass* = *parent.sdldefinition*$^{-1}$ – if *parentclass* is empty, set *parentclass* = *package.systemclass*

Rule 191. Translate Context for New Block

Precondition	– *block* is not a block instance of a block type declared within *system*$_{new}$ – *block.name* ≠ "processesblock" – *block.processesblock*$^{-1}$ is empty – *block.name* ≠ *parent.name* } Check that *block* is not a processblock. – *package* does not contain a class with name *block.name*
Action	Add a new class *class* to *package* with the following properties – *class.name* = *block.name* – *class.stereotype* = «block» – If *block* is a block type or block instance: *class.typed* = true – If *block* is a block definition: *class.typed* = false – If *block* is block instance and corresponding block type is not visible: *block.extern* = true – If *block* is a block type or block definition: *class.definedIn* = *parentclass*
Variables	– *newclass* is the newly created class – *class.sdldefinition* = *block*

Rule 192. Translate New Block

Precondition	*block* is not a block type (it is a block instance or a block definition)
Action	Add a new *aggregation* to *package* with the following properties – *aggregation.aggergate = parentclass* – *aggregation.component = newclass* – *aggregation.componentrole = block.name* – *aggregation.composite* = true
Variables	– *aggregation.sdlcomponent = block*

Rule 193. Aggregate for Block Definition and Instance

V 8.5 Delete Block

A naïve approach would be to simply delete the class that is linked with deleted block. However, we have to check whether the class at issue is not linked with other information, more specifically, a management process or a processes block. An important condition that must be fulfilled is to find a class that is linked with the deleted class as the *sdldefinition*, see Rule 194. If the delete block is a processes block or an architecture block, there is no such link. Consequently, nothing changes in the UML model. Rule 197 defines the case where the linked class is also linked with a process. Instead of deleting the class, the class is transformed into a «process» class.

Context for this section	– $block_{old}$ is the deleted SDL block, block type or block instance – $class = block_{old}.sdldefinition^{-1}$

Rule 194. Translate Context for Delete Block

Precondition	– $class \neq \text{empty} \wedge class \neq package.systemclass$ – $block_{old}$ is a block type or block definition – *class.sdlprocess* is empty or is a process instance
Action	Delete *class* in $package_{new}$.

Rule 195. Deleted Standard Block

Precondition	– $block_{old}$ is a block instance
Context	– $aggregation = block_{old}.sdlcomponent^{-1}$
Action	Delete *aggregation* from *package*

Rule 196. Deleted Block Instance

Precondition	– $class \neq \text{empty} \wedge class \neq package.systemclass$ – $block_{old}$ is a block type or block definition

	– *class.sdlprocess* is a process definition or process type
Action	Change the following properties of $class_{new}$ – *class.stereotype* = «process» – *class.type* = false – *class.definedIn = class.sdlprocess.parent.sdldefinition*$^{-1}$

Rule 197. Transform «block» class into «process» class

V 8.6 Compare Block

Changes in a block are only translated if it is the main link (*sdldefinition*) of a class. Changes in the processes blocks, architecture block or block that are not linked can safely be ignored, see Rule 199. The rules below check whether the block has been renamed or moved to a different place, became (non-)type, has been switched from instance to definition or the type name of an instance has changed. The translation of changing the specialization in defined in section V 8.10.

Context for this section	– $block_{old}$ is the old SDL block (or block type or block instance) – $block_{new}$ is the new SDL block to be compared with $block_{old}$ – *class* = $block_{old}$*.sdldefinition*$^{-1}$ – $parent_{old}$ = the SDL structure that surrounds $block_{old}$ – $parent_{new}$ = the SDL structure that surrounds $block_{new}$

Rule 198. Translate Context for Compare Block

Precondition	– *class* is empty
Action	Do nothing; do not check the rules below.

Rule 199. Skip Unlinked Block

Precondition	– $block_{old}$*.name* ≠ $block_{new}$*.name*
Action	Set *class.name* = $block_{new}$*.name*

Rule 200. Translate Block Rename

Rule 201 translates the case where the block is moved to a different structure. For block types, this is translated by adapting the *definedIn* link of the class. For block definition and block instances, this is translated by modifying the aggregation that represents the block. The aggregate end of the association is modified to point to the class that is linked with the new aggregate structure.

Precondition	– $parent_{old}$ ≠ $parent_{new}$ – $parent_{new}$*.sdldefinition*$^{-1}$ is not empty
Context	– *aggregation* = $block_{old}$*.sdlcomponent*$^{-1}$ – $parentclass_{new}$ = $parent_{new}$*.sdldefinition*$^{-1}$

Action	If $\underline{block_{new}}$ is typed, set *class.definedIn = parentclass*$_{new}$ else, set *aggregation.aggregate = parentclass*$_{new}$

Rule 201. Translate Block Move

Rule 202 and Rule 203 translates the switch between block type and a non-typed block. Except for changing the *type* attribute of the class, the aggregation that is used for non-typed classes (≈block definition) must be added or deleted. In the case that the block type becomes a block definition, we assume that the instances of the old block type are deleted and consequently the aggregations that represent the instances are deleted too.

Precondition	– $\underline{block_{old}}$ is a block definition – $\underline{block_{new}}$ is a block type
Context	– *aggregation* = $\underline{block_{old}}$*.sdlcomponent*$^{-1}$
Action	Set *class.type* = true. Delete *aggregation* from *package.*

Rule 202. Block Becomes Typed

Precondition	– $\underline{block_{old}}$ is a block type – $\underline{block_{new}}$ is a block definition
Context	– *parentclass*$_{new}$ = $\underline{parent_{new}}$*.sdldefinition*$^{-1}$
Action	Set *class.type* = false. Add an *aggregation* with the following properties – *aggregation.aggergate = parentclass*$_{new}$ – *aggregation.component = class* – *aggregation.composite* = true – *aggregation.*$\underline{sdlcomponent}$ = $\underline{block_{new}}$

Rule 203. Block Becomes Untype

Precondition	– $\underline{block_{old}}$ is not a block instance and $\underline{block_{new}}$ is a block instance **or** – $\underline{block_{old}}$ is a block instance and $\underline{block_{new}}$ is not a block instance
Action	Translate $\underline{block_{old}}$ as a deleted block and $\underline{block_{new}}$ as a new block.

Rule 204. Switch Instance – Non-Instance

Rule 205 translates the case where the type of a block instance is modified. In other words, it is a different type from which an instance is defined. In terms of UML, this means that the component role of the aggregation must be moved to a different type.

Precondition	– $\underline{block_{old}}$ and $\underline{block_{new}}$ are both block instances – $\underline{block_{old}.type} \neq \underline{block_{new}.type}$
Context	– $aggregate = \underline{block_{old}}.sdlcomponent^{-1}$ – $typeclass_{new} = \underline{block_{new}.type}.sdldefinition^{-1}$ – $typeclass_{old} = \underline{block_{old}.type}.sdldefinition^{-1}$
Action	If $typeclass_{new}$ is empty or $typeclass_{old}$ is empty – Translate $\underline{block_{old}}$ as a deleted block instance. – Translate $\underline{block_{new}}$ as a new block instance. Else – $aggregate.component = typeclass_{new}$ – $aggregate.\underline{sdlcomponent} = \underline{block_{new}}$

Rule 205. Translate Type Change of Block Instance

V 8.7 New Process

If the new process is an independent process, it is translated into a new class with the same name (Rule 208). The new process is considered dependend of it is the only class in a block or if the process has the same name as the parent or the parent's parent. In that case, the process is considered to be the *management process* of the block and the link is set accordingly (Rule 207). In addition to the creation of the class, Rule 209 creates an aggregation for process instances and process definitions.

Context for this section	– *process* is the new SDL process, process type or process instance – *parent* is the surrounding SDL block (type) of *process* – *parentclass* = *parent*. *processesblock* $^{-1}$ – if *parentclass* is empty, let = *parent*. *sdldefinition* $^{-1}$ – if *parentclass* is empty, set *parentclass* = *package.systemclass*

Rule 206. Translate Context for New Process

Precondition	– *process.name* = *parent.name* **or** – *process.name* = *parentclass.name* **or** – *process* is the only process in *parent* and *parentclass.processesblock* = *parent*
Action	Set *parentclass.managementprocess* = *process*. Stop comparing this process.

Rule 207. Translate New Management Process

Precondition	– *process* is not a process instance **or** *process* is a process instance of a type of another package (external) – *package* does not contain a class with name *process.name*

Action	Add a new class *class* to *package* with the following properties – *class.name* = *process.name* – *class.stereotype* = «process» – If *process* is a process type or process instance: *class.typed* = true – If *process* is a process definition: *class.typed* = false – If *process* is process instance and corresponding process type is not visible: *class.extern* = true – If *process* is a process type or process definition: *class.definedIn* = *parentclass*
Variables	– *newclass* is the newly created class – *class.sdldefinition* = *process*

Rule 208. Translate New Process

Precondition	*process* is not a process type (it is a process instance or a process definition)
Action	Add a new *aggregation* to *package* with the following properties – *aggregation.aggergate* = *parentclass* – *aggregation.component* = *newclass* – *aggregation.componentrole* = *process.name* – *aggregation.composite* = true – *aggregation.sdlcomponent* = *process*

Rule 209. Aggregate for Process Definition and Instance

V 8.8 Delete Process

The deleted process can be linked with a class in two ways. If the process is the main link (*sdldefinition*), then the class is deleted (Rule 211). If the process is only the management process of the class, nothing happens (no rule). When the process is a process instance, the process is linked with a aggregation and so the aggregation is deleted (Rule 212).

Context for this section	– $process_{old}$ is the deleted SDL process, process type or process instance – $class = process_{old}.sdldefinition^{-1}$

Rule 210. Translate Context for Delete Block

Precondition	– *class* ≠ empty – $process_{old}$ is a process type or process definition
Action	Delete *class* in $package_{new}$.

Rule 211. Deleted Standard Process

Precondition	– $\underline{process_{old}}$ is a process instance
Context	– $aggregation = \underline{process_{old}}.sdlcomponent^{-1}$
Action	Delete *aggregation* from *package*

Rule 212. Deleted Process Instance

V 8.9 Compare Process

Similar to comparing blocks, the properties of a process that are compared and translated, are its name (rename), its scope (move), switch between type and non-typed or instance and non-instance. However, most of the rules are only fired if the the process is the main link of the class; in other words, if the process is not just the management process of a class.

Context for this section	– $\underline{process_{old}}$ is the old SDL process (or process type or process instance) – $\underline{process_{new}}$ is the new SDL process to be compared with $\underline{process_{old}}$ – $class = \underline{process_{old}}.sdldefinition^{-1}$ – $processclass = \underline{process_{old}}.sdlprocess^{-1}$ – $aggregation = \underline{process_{old}}.sdlcomponent^{-1}$ – $\underline{parent_{old}}$ = the SDL block (type) that surrounds $process_{old}$ – $\underline{parent_{new}}$ = the SDL block (type) that surrounds $process_{new}$

Rule 213. Translate Context for Compare Process

Precondition	– $\underline{process_{old}}.name \neq \underline{process_{new}}.name$ – $class \neq$ empty
Action	Set $class.name = \underline{process_{new}}.name$

Rule 214. Translate Process Rename

Precondition	– $\underline{process_{old}}.name \neq \underline{process_{new}}.name$ – $aggregate \neq$ empty
Action	Set $aggregate.componentrole = \underline{process_{new}}.name$

Rule 215. Translate Process Instance Rename

Precondition	– $\underline{process_{old}}$ is not a process instance and $\underline{process_{new}}$ is a process instance **or** – $\underline{process_{old}}$ is a process instance and $\underline{process_{new}}$ is not a process instance
Action	Translate $\underline{process_{old}}$ as a deleted process and $\underline{process_{new}}$ as a new process.

Rule 216. Switch Instance – Non-Instance

Rule 217 specifies the translation of a process that is moved to a different scope. This is achieved by using the *processeblock* back link. Note that, if there is no explicit processes block, $processesblock^{-1}$ still returns the correct class.

Precondition	– $\underline{parent}_{old} \neq \underline{parent}_{new}$ – *class* ≠ empty
Context	– $parentclass_{new} = \underline{parent}_{new}.processesblock^{-1}$
Action	If $parentclass_{new}$ is not empty, set $aggregate.aggregate = parentclass_{new}$.

Rule 217. Translate Process Move

Rule 218 and Rule 219 are the process equivalent of Rule 202 and Rule 203. They translate a conversion from process type to process and the other way around.

Precondition	– $\underline{process}_{old}$ is a process definition – $\underline{process}_{new}$ is a process type
Action	Set *class.type* = true. *class.definedin = aggregation.aggregate.* Delete *aggregation* from *package.*

Rule 218. Process becomes Typed

Precondition	– $\underline{process}_{old}$ is a process type – $\underline{process}_{new}$ is a process definition
Action	Set *class.type* = false. Add an aggegation *aggregation* with the following properties – $aggregation.aggergate = parentclass_{new}$ – *aggregation.component = class* – *aggregation.composite* = true

Rule 219. Process becomes Untype

Rule 220 translates the case where a process instance is changed to a different type. Note that $\underline{process.type}$ points to a SDL process type, in contrast with the *class.type* that is a boolean. The if part in the action section deals with the case when one of the necessary links are broken.

Precondition	– $\underline{process}_{old}$ and $\underline{process}_{new}$ are both process instances – $\underline{process}_{old}.\underline{type} \neq \underline{process}_{new}.\underline{type}$
Context	– $aggregate = \underline{process}_{old}.sdlcomponent^{-1}$ – $typeclass_{new} = \underline{process}_{new}.\underline{type}.sdldefinition^{-1}$ – $typeclass_{old} = \underline{process}_{old}.\underline{type}.sdldefinition^{-1}$

Action	If one of $typeclass_{new,}$ $typeclass_{old}$ or *aggregate* are empty – Translate $\underline{process_{old}}$ as a deleted process instance. – Translate $\underline{process_{new}}$ as a new process instance. Else – *aggregate.component* = $typeclass_{new}$ – *aggregate.*$\underline{sdlcomponent}$ = $\underline{process_{new}}$

Rule 220. Process Instance of different Type

V 8.10 New/Delete/Compare Specialization

In SDL, a specialization is not a real entrity on its own, it's rather an attribute of a block type or a process type. Nevertheless, we translate the specialization separately because the translation is common for processes and blocks. Rule 221 is only fired if the comparison engine found a difference in the specialization of a structured typed, therefore there is no precondition to the rule.

Context	– *class* = $\underline{struct_{new}}$*.sdldefinition*$^{-1}$ – $\underline{struct_{new}}$ is the SDL process or block type with the changed specialization – $\underline{superstruct_{new}}$ = $\underline{struct_{new}}$*.specialization* (possibly empty) – *superclass* = $\underline{superstruct_{new}}$*.sdldefinition*$^{-1}$ (possibly empty)
Action	– *class.superclass* = *superclass*

Rule 221. Translate Context for New Specialization

V 8.11 New Procedure

A new SDL procedure added to a process is translated by adding an UML procedure to the class that is linked with the process.

Context	– is the new SDL procedure – $\underline{process}$ is the process containing $\underline{procedure}$ – *class* = $\underline{process}$*.sdlprocess* $^{-1}$ – $\underline{procedure.params}$ = ($\underline{p_1}$, …, $\underline{p_n}$)
Action	Add an new operation *oper* to *class* with the following attributes: – *oper.name* = $\underline{procedure}$*.name* – *oper.returntype* = $\underline{procedure}$*.returns* – ∀ i ∈ 1..n : add a parameter to *oper*, with name $\underline{p_i.variable}$ and type $\underline{p_i.type}$.

Rule 222. Translate New Procedure

V 8.12 Delete Procedure

Context	– $\underline{procedure}_{old}$ is the deleted SDL procedure – $operation = \underline{procedure}_{old}.sdlprocedure^{-1}$
Action	Delete *operation* from its class.

Rule 223. Translate Deleted Procedure

V 8.13 Compare Procedure

For a procedure, the name, parameters and return sort are compared. We differentiate between two kinds of changes in the parameter list. *Rule* 226 is fired if there are more or less parameters or if the name of one of the parameters is changed. *Rule* 227 is fired if only the sort of one or more of the parameters had changed.

Context for this section	– $\underline{procedure}_{old}$ is the old SDL procedure – $\underline{procedure}_{old}.parameters = (\underline{p}_{1,old}, \ldots, \underline{p}_{n,old})$ – $\underline{procedure}_{new}$ is the new SDL procedure – $\underline{procedure}_{new}.parameters = (\underline{p}_{1,new}, \ldots, \underline{p}_{m,new})$ – $operation = \underline{procedure}_{old}.sdlprocedure^{-1}$

Rule 224. Translate Context for Delete Procedure

Precondition	– $\underline{procedure}_{old}.name \neq \underline{procedure}_{new}.name$
Action	Set $operation.name = \underline{procedure}_{new}.name$

Rule 225. Translate Procedure Rename

Precondition	– $n \neq m$ **or** – $\exists\, i \in (1..n) : \underline{p}_{i,old}.name \neq \underline{p}_{i,new}.name$
Action	Delete all the parameters in *operation* and regenerate the parameters as defined in **Error! Reference source not found.**. Skip *Rule* 227.

***Rule* 226. Translate Parameter Changes**

Precondition	– $n = m$ Ù $\forall\, I \in (1..n) : \underline{p}_{i,old}.name \neq \underline{p}_{i,new}.name$ – $\exists\, j \in (1..n) : \underline{p}_{i,old}.type \neq \underline{p}_{i,new}.type$
Action	Set the type of the j_{th} parameter to $\underline{p}_{i,new}.type$ Repeat for other parameters if necessary.

***Rule* 227. Translate Parameter Type Change**

Precondition	– $\underline{procedure}_{old}.returns \neq \underline{procedure}_{new}.returns$

Action	Set *operation.returntype* = $\underline{procedure_{new}}$*.returns*

***Rule* 228. Translate Procedure Rename**

V 8.14 Communication

The reverse incremental translation of communication routes into associations is somewhat problematic. In general, there is no one-on-one link between an association and a channel or signal route. In most cases, an association is linked with two or more channels, signal routes and/or gates. If one of these parts is renamed or deleted, the original association is not necessarily affected. Most of the support for communication is therefore focused on forward engineering instead of reverse engineering. Nevertheless, we do provide reverse translation of changes in some specific cases. More specifically, we translate a signal routes and channels if it does not go to the environment. We translate a deleted route if it was the last route linked with association.

V 8.15 New Communication

Context for this section	– *route* is the new SDL signal route or channel

Rule 229. Context for new Communication

Precondition	– *route.fromstruct.parent* = *route.tostruct.parent*
Context	– *fromclass* = *route.fromstruct.*$sdldefinition^{-1}$ – *toclass* = *route.tostruct.*$sdldefinition^{-1}$
Action	Create a new association *assoc* with the following attributes – *assoc.name* = *route.name* – *assoc.stereotype* = «communication» – *assoc.fromclass* = *fromclass* – *assoc.toclass* = *toclass* – *assoc.channels* = {*route*}

Rule 230. Create Association for Full Route

V 8.16 Delete Communication

Context for this section	– *route* is the deleted SDL signal route/channel – *association* = *route*. $sdlchannel^{-1}$

Rule 231. Context for Deleted Communication

Action	Remove *route* from the association.*sdlchannels* list

Rule 232. Delete link to Deleted Route

Precondition	– *association.sdlchannels* is empty
Action	Delete *association*

Rule 233. Delete Unlinked Association

V 8.17 Compare Communication

Communication routes (signal routes & channels) have several attributes that can change, but only in specific cases this results in a change on linked UML association. A route rename (Rule 235) is only translated if the routes linked at both ends of the association have the same name. A route that is connected to a different structure is translated by reconnecting the association to a different class (Rule 236 and Rule 237), only if the route represents an association end.

Context for this section	– $route_{old}$ is the old SDL signal route/channel – $route_{new}$ is the new SDL signal route/channel to be compared with $route_{old}$ – *association* = $route_{old}.sdlfrom^{-1}$ – if *association* is empty, *association* = $route_{old}.sdlto^{-1}$ – if *association* is empty, *association* = $route_{old}.sdlchannel^{-1}$
Precondition for this section	– *association* is not empty

***Rule* 234. Translate Context for Comparing Communication Routes**

Precondition	– $route_{old}.name \neq route_{new}.name$ – *association.sdlfromroute.name = association.sdltoroute.name* – *association.sdlfromchannel.name = association.sdltochannel.name*
Action	*association.name* = $route_{new}.name$

Rule 235. Translate Route Rename

Precondition	– $route_{old}$ = *association.sdlfromroute* **or** $route_{old}$ = *association.sdlfromchannel* – $route_{old}.fromstruct \neq route_{new}.fromstruct$
Action	If $route_{new}.fromstruct$ is a block, *association.fromclass* = $route_{new}.fromstruct.sdldefinition^{-1}$ Else *association.fromclass* = $route_{new}.fromstruct.sdlprocess^{-1}$

Rule 236. Translate Route From Destination Change

Precondition	– $route_{old}$ = *association.sdltoroute* or $route_{old}$ = *association.sdltochannel* – $route_{old}.tostruct \neq route_{new}.tostruct$

Action	If $\underline{route_{new}}.tostruct$ is a block, $association.toclass = \underline{route_{new}}.tostruct.sdldefinition^{-1}$ Else $association.toclass = \underline{route_{new}}.tostruct.sdlprocess^{-1}$

Rule 237. Translate Route To Destination Change

V 8.18 New newtype

A new SDL newtype declaration is translated by creating a class with stereotype «newtype». The context variables defined in Rule 238 imediately parse the signature and behaviour into attributes and operators. For each attribute or operator, respectively Rule 240 or Rule 241 is executed to translated them into class attributes and operations.

Context for this section	– $\underline{newtype}$ is the new SDL newtype If $\underline{newtype}$ is a "struct" newtype, let : – $\underline{newtype.signature} = (\underline{attr_1}, \ldots, \underline{attr_n})$ is the list of attributes – $\underline{attr_i} = (name_i, type_i)$ – $\underline{newtype.behaviour} = (\underline{oper_1}, \ldots, \underline{oper_m})$ is the list of operators – $\underline{oper_j} = (name_j, parameters_j, returntype_j)$

Rule 238. Translate Context for New Newtype

Action	Create a new class *class* with the following properties: – $class.name = \underline{newtype}.name$ – $class.stereotype$ = «newtype»
Variables	*class* is the newly created class

Rule 239. Create «newtype» class

Context	$\forall i \in (1..n)$: execute the action with $\underline{attr_i}$
Action	Add a new attribute *attr* to *class* with the following properties: – $attr.name = \underline{attr_i}.name$ – $attr.type = \underline{attr_i}.type$

Rule 240. Translate Attribute in Newtype

Context	$\forall i \in (1..m)$: fire this rule with $\underline{oper_j}$ Let $\underline{parameters_j} = (partype_1, \ldots, partype_n)$ a list of Strings denoting types.
Action	Add a new operation *oper* to *class* with the following properties: – $oper.stereotype$ = «operator» – $oper.name = \underline{oper_j}.name$

	– *oper.returntype* = $\underline{attr_j}$*.returntype* – ∀ j ∈ (1..n) : add a parameter *parameter* to *oper* with the following properties: *parameter.name* = "par<j>", *parameter.type* = $partype_j$

Rule 241. Translate Operator in Newtype

V 8.19 Delete newtype

Deleting a newtype has only an effect on the UML model if the newtype is linked with a class. In other words, deleting the newtypes generated from types and return types, is not translated back to UML.

Context	– $\underline{newtype_{old}}$ is the deleted SDL new type – *class* = $\underline{newtype_{old}}$*.sdldatatype*$^{-1}$
Action	If *class* is not empty, delete *class*.

Rule 242. Translate Deleted New Type

V 8.20 Compare newtype

The newtype comparison checks all components separately: the name, the attributes and the operators. However, we cannot translate renamings of individual attributes or operators.

Context for this section	– $\underline{newtype_{old}}$ is the old SDL new type – $\underline{newtype_{old}}$*.attributes* = ($\underline{a}_{1,old}$, …, $\underline{a}_{n,old}$) – $\underline{newtype_{new}}$*.operators* = ($\underline{p}_{1,old}$, …, $\underline{p}_{m,old}$) – $\underline{newtype_{new}}$ is the new SDL new type – $\underline{newtype_{new}}$*.attributes* = ($\underline{a}_{1,new}$, …, $\underline{a}_{n,new}$) – $\underline{newtype_{new}}$*.operators* = ($\underline{p}_{1,new}$, …, $\underline{p}_{m,new}$) – *operation* = $\underline{newtype_{old}}$*.sdlnewtype*$^{-1}$

Rule 243. Translate Context for Delete New Type

Precondition	– $\underline{newtype_{old}}$*.name* ≠ $\underline{newtype_{new}}$*.name*
Action	Set *operation.name* = $\underline{newtype_{new}}$*.name*

Rule 244. Translate New Type Rename

As the attributes of newtype do not have an own identity in the information model, it is not possible to detect individual renames. If the number of attributes and their names are the same, then the types of the attributes are compared, otherwise all the attributes are regenerated. The same approach is taken for comparing the operators of the newtype.

Precondition	– n,old ≠ n,new **or**

	– $\exists$ i $\in$ (1..n) : $\underline{a}_{i,old}.name \neq \underline{a}_{i,new}.name$
Action	Delete all the attributes in *class* and regenerate the attributes as defined in Rule 240.

Rule 245. Translate Attribute Changes

Precondition	– n,old = n,new Ù $\forall$ i $\in$ (1..n) : $\underline{a}_{i,old}.name = \underline{a}_{i,new}.name$ – $\forall$ j $\in$ (1..n) : $\underline{p}_{i,old}.type \neq \underline{p}_{i,new}.type$: execute the action
Action	Set the type of the j_{th} attribute to $\underline{p}_{i,new}.type$

Rule 246. Translate Attribute Type Change

Precondition	– m,old $\neq$ m,new **or** – $\exists$ i $\in$ (1..m) : $\underline{p}_{i,old}.name \neq \underline{p}_{i,new}.name$
Action	Delete all the operations in *class* and regenerate the operations as defined in Rule 241.

Rule 247. Translate Operator Changes

Precondition	– m,old = m,new Ù $\forall$ i $\in$ (1..n) : $\underline{p}_{i,old}.name = \underline{p}_{i,new}.name$ – $\exists$ j $\in$ (1..m) : $\underline{p}_{i,old}.returntype \neq \underline{p}_{i,new}.returntype$
Action	Set the type of the j_{th} operation to $\underline{p}_{i,new}.returntype$

Rule 248. Translate Parameter Type Change

V 8.21 New Variable

In SDL, attributes can only appear in processes, so it is easy to find the correct class to translate the SDL variable into a UML attribute.

Context	– *variable* is the new SDL variable – *process* is the SDL process that contains *variable* – *class* = *parent*. $sdlprocess^{-1}$
Action	Add a new attribute *attribute* to *class* with the following properties: – *attribute.name* = *variable.name* – *attribute.type* = *variable.type* – *attribute.default* = *variable.initialexpr* – *attribute.sdldeclaration* = *variable*

Rule 249. Translate New Variable

V 8.22 Delete Variable

Context	– *variable* is the deleted SDL variable – *attribute* = *variable*. *sdldeclaration* $^{-1}$
Action	Delete *attribute* from its class.

Rule 250. Translate Deleted Variable

V 8.23 Compare Variable

Here, all the properties of an SDL variable are compared and translated if necessary: name, type and default value.

Context for this section	– $variable_{old}$ is the old SDL variable – $variable_{new}$ is the new SDL variable – *attribute* = $variable_{old}$. *sdldeclaration* $^{-1}$

Rule 251. Translate Context for Delete Variable

Precondition	– $variable_{old}.name \neq variable_{new}.name$
Action	Set $attribute.name = variable_{new}.name$

***Rule* 252. Translate Variable Rename**

Precondition	– $variable_{old}.type \neq variable_{new}.type$
Action	Set $attribute.type = variable_{new}.type$

Rule 253. Translate Variable Type Change

Precondition	– $variable_{old}.initialexpr \neq variable_{new}.initialexpr$
Action	Set $attribute.default = variable_{new}.initialexpr$

Rule 254. Translate Variable Initial Expression Change

V 8.24 New State

The SDL state and UML state (after flattening) map one-on-one and has only one property, its name. Therefore, translating a new state, delete state or renamed state is straightforward.

Context	– *state* is the new SDL state – *process* is the SDL process that contains *state* – *class* = *parent*. *sdlprocess* $^{-1}$ – *statediagram* = *class.statediagram*

Action	Add a new state *state* to *statediagram* with the following properties: – *state.name* = *state.name* – *state.type* = normal

Rule 255. Translate New State

V 8.25 Delete State

Context	– *state* is the deleted SDL state – *state* = *state*. $sdlstate^{-1}$
Action	Delete *state* from its state diagram.

Rule 256. Translate Deleted State

V 8.26 Change State

Context for this section	– $state_{old}$ is the old SDL state – $state_{new}$ is the new SDL state – *state* = *state*. $sdlstate^{-1}$

Rule 257. Translate Context for Delete State

Precondition	– $state_{old}.name \neq state_{new}.name$
Action	Set $state.name = state_{new}.name$

Rule 258. Translate State Rename

V 8.27 New Transition

The transition concepts are somewhat different in SDL and UML. In SDL, a transition is identified by its input and/or guard. In UML, a transition has its own identity and the input and guard are properties. Moreover, a SDL transition can split into different transitions with different destination states, while in UML the transition has one fixed destination state. In the context definition in Rule 259, we make a list of all the destination states of the transition. For each of the destinations, Rule 261 creates another transition in UML. Rule 260 handles the case where the new tansition is actually the start transition (*sdlprocess.start*). In that case a new start state is created in the UML state diagram.

Context for this section	– *transition* is the new SDL transition – *source* is the source states of *transition* – *source* = $source.state^{-1}$ – $\{dest_1, \ldots, dest_n\}$ are all the possible destination states of *transition* – $\forall j \in (1..n)$: execute rule Rule 261 with *deststate* = $dest_j$

	– let *process* be the process containing *transition* – *statediagram* = *process.sdlprocess*$^{-1}$*.statediagram*

Rule 259. Translation Context for New Transition

Precondition	– *transition* = *process.start*
Action	Let *source* be the start state in *statediagram*. If *source* is empty, add a new state *source* to *statediagram* with the following properties: – *source.type* = start – *source.name* = ""

Rule 260. Translate New Start Transition

Context	– *deststate* is the added destination for *transition* – *nextstate* is the next state action of *transition* that correspond with *deststate*
Action	Add a new transition *transition* to *statediagram* with the following properties: – *transition.name* = *transition.input* (may be empty) – *transition.guard* = *transition.enable* (may be empty) – *transition.source* = *source* – *transition.dest* = $dest_i.state^{-1}$
Variables	– *transition.nextstate* = *nextstate* – *transition.sdltransition* = *transition*

Rule 261. Translate New Transition Destination

V 8.28 Delete Transition

Context	– $transition_{old}$ is the SDL transition with deleted destinations – *deststate* is the deleted destination state from $transition_{old}$ – *state* is the source state of $transition_{old}$ – *nextstate* is the nextstate action of *transition* that correspond with *deststate* – *transition* = *nextstate.nextstate*$^{-1}$
Action	Delete the UML *transition* from its state diagram.

Rule 262. Translate Context for Deleted Transition

V 8.29 Compare Transition

Context for this section	– $transition_{old}$ is the old SDL transition – $transition_{new}$ is the new SDL transition to be compared with $transition_{old}$

	– *deststate* is the destination state of $transition_{old}$ and $transition_{new}$ – *state* is the source state of $transition_{old}$ and $transition_{new}$ – *nextstate* is the nextstate action of $transition_{old}$ that correspond with *deststate* – $transition = nextstate.nextstate^{-1}$

Rule 263. Translate Context for Deleted Transition

Precondition	– $transition_{old}.input \neq transition_{new}.input$
Action	Set $transition.event = transition_{new}.input$

Rule 264. Translate Transition Rename

Precondition	– $transition_{old}.enable \neq transition_{new}.enable$
Action	Set $transition.guard = transition_{new}.enable$

Rule 265. Translate Transition Type Change

V 8.30 New Action

Context for this section	– *action* is the new SDL action – *transition* is the transition containing action – {$nextstate_1$, …, $nextstate_n$} are all the possible nextstate statements reachable from action. – {$trans_1$, …, $trans_n$} are the corresponding UML transitions ($trans_i = nextstate_i.nextstate^{-1}$)

Rule 266. Translate Context for Deleted Transition

Precondition	– *action* is an signal output action
Context	– Parse *action* as: <signal> <parameters> TO <destination> Leave destination empty if action does not contain "TO"
Action	∀ j ∈ (1..n) : add the action to $trans_j$: `;^<destination>.<signal><parameters>` **or** `;^<signal><parameters>` if <destination> is empty

Rule 267. Translate New Output Action

Precondition	– *action* is an assignment
Context	– Parse *action* as: TASK <assignment>;

Action	∀ j ∈ (1..n) : add the action to $trans_j$: `<assignment>;`

Rule 268. Translate New Assignment

Precondition	– *action* is a informal task
Context	– Parse *action* as: TASK '<task>';
Action	∀ j ∈ (1..n) : add the action to $trans_j$: `; <task>`

Rule 269. Translate New Informal Task

Precondition	– *action* is not a signal output nor task.
Action	∀ j ∈ (1..n) : add the action to $trans_j$: `; action`

Rule 270. Translate New General Action

V. 9 UML Post Processing

V 9.1 Pass changes on to full UML model

Unlike during the forward iteration, the resulting (UML) model cannot simply be stored with the necessary details to obtain the updated system design model. The internal model has been preprocessed by adding and changing information that is undesirable in a design document. Therefore, the changes applied to the internal information model are reapplied to the original UML model. There are two approaches to realize the forwarding of changes.

One option is to keep a parallel data structure that is not preprocessed during the whole iteration process. Every time a change is translated to UML, the change is also immediately applied to the parallel data structure. If this parallel data structure in addition contains all other UML model information (use cases, sequence diagrams, etc.), it is sufficient to store the model as the new system design version.

The second option is to keep a copy of the model directly after preprocessing. After the incremental translation, this copy is compared with the model that has been updated to discover the changes made by the translator. The UML translator used during the forward iteration can be reused for this purpose. Next, these changes are applied to the full version of the UML model. This approach has the major advantage that it can be applied to any proprietary data structure with an API. Moreover, the UML compare operation is already available because it is necessary for the forward iteration.

V 9.2 Create and Update Diagrams

Besides updating the UML model, the diagrams that visualize the information in the model also need to be updated. Entities that were renamed or deleted are automatically reflected in the existing diagrams. Also new attributes and operations are automatically shown if the class at hand is present in a diagram. However, new classes and their contents and new class relationships are shown nowhere. Unfortunately, there is no universal way to arrange the new information into diagrams, as it depends, among others, on the semantics of the model. The generation of extra diagrams is of particular interest when a significant system design effort has been performed in SDL. Therefore, the best approach is to let the user choose a number of diagrams out of a set of possible diagrams that should be generated. A non-exhaustive list of possible diagrams is:

- All new classes in the whole model and the relationships between the new classes.
- The new classes per package and their relationships.
- Diagram for each new class and its related classes, with their relationships.
- All new generalization relationships with the corresponding classes showing the generalization hierarchy, possibly including the existing super classes.
- All new aggregation relationships with the corresponding classes showing the aggregation tree, possibly including existing aggregate classes.

- All new associations relationships with the corresponding classes showing the communication graph.
- All classes that have new attributes and/or new operations.

V. 10 User Interaction

At this point, we have a large set of translation rules of how changes are translated. When we consider a tool that implement these rules, we do not want an all or nothing approach. Even more than setting a number of global options to influence the incremental translator, the user need to have a more control over how changes are interpreted, translated and applied. In this section, we describe a number of mechanisms that allows the user to fine-tune the incremental translation. Sections V 10.1 through V 10.3 discuss mechanisms that are applied dynamically during the incremental translation and require direct user interaction. Sections V 10.4 through V 10.6 discuss static ways to control the incremental translation process: exclude certain parts of the model or specification for comparison or modification, allow a maintenance phase during which changes are not registered and the generation of a change report. A particular tool may support a selection out of these dynamic and static techniques, but probably not all of them as their functionality overlap partially.

V 10.1 Interactive Comparison

The first technique allows the user to confirm or cancel the detected changes before translation. The first step in the incremental translation is detecting the changes by comparing the previous version and the most recent version of the model or specification. For a number of reasons, the comparison may fail to find some changes or may detect a change that is actually unintentional or not relevant. For example, the UML comparison notices a new class and a deleted class, while actually the user intended to rename the class. Another example is when in SDL two transitions are merged into one by using a decision. The second transition will be marked as deleted, but that was not the intention.

After the comparison, the change list is presented to the user, who can then discard incorrect changes or merge a delete and new change into a comparison. To make the evaluation easier, the user can view the deleted, new or modified entity in its context.

Even if a structural editor is used and all changes are recorded immediately, this technique can still be applied.

V 10.2 Interactive Rule Activation

The second technique is to make the execution of rules during the translation process interactive. During the incremental translation, the detected changes are translated by firing the corresponding rules in a certain context. With this approach, to user confirms the execution of each rules or group of rules before it is fired. To ease the decision, the UML or SDL context to which the rule applies is shown, with or without the rule executed. Answers like “always yes”, “always yes for this class” and “always no” avoid repetitive decisions.

V 10.3 Managing Links

The third technique is to manually restore missing links between UML and SDL. The UML to SDL links are crucial for the incremental translation. Modification to an unlinked entity cannot be translated as there is no scope to translate that change to. In certain circumstances links may disappear, for example when an entity is deleted and remodeled or when some translation rules were not executed. For this reason, the user should be able to fill in missing links during the translation as well as during modeling. Actually, current tools already support linking entities between different abstraction layers. For example, Telelogic Tau describes three ways to create links (implinks) as support for its SOMT method [And95]: manually, by linking together two endpoints or by linking together an endpoint and a selected object or automatically, by copying and pasting an object (Paste As). Although implinks differ from our links, the tool support is very similar. Implinks can link any UML entity with any SDL entity, while our hierarchical UML-SDL links used for round-trip engineering have more constraints, i.e. each link must be of a certain type. Moreover, some entity have several distinct links, e.g. a class can be linked with a block (*sdldefinition*) and a process (*sdlprocess*) at the same time. The only extra tool support to adapt implinks to hierarchical links is an extra option to choose which link is specified.

V 10.4 Protect Areas

With this technique, the user can make an UML or SDL entity read and/or write protected. If an entity is protected, its sub-entities are protected too. A write-protected entity is not updated during the incremental translation. For example, if an SDL process is write-protected and in the linked UML state diagram a state is deleted, this state will not be deleted from the process. A read protected entity is not taken into account during the comparison. For example, if an SDL process is read protected, non of the changes done to the process or to its contents are processed nor translated.

A typical use of this technique is to make a analysis-only UML package read protected, such that it is not translated and changes in the package are ignored. An SDL process can be made write protected after testing to avoid accidental changes while editing or updating the UML model. On a smaller scale, a transition in a UML state diagram with actions in general terms, e.g. ‘process data’, can be made write protected to avoid that it becomes overloaded with all the actions added during the implementation in SDL.

V 10.5 Maintenance Phase

The maintenance phase allows the developer to correct the updated model after an incremental translation. This phase takes place after the translation process is finished and before the normal system design or detailed design is resumed. During the maintenance phase, the changes made to the UML model or to the SDL specification are ignored. The idea is to clean up unwanted artifacts of the translation process, without adding new information. It can be seen as a manual extension of the incremental translation. The resulting model or specification after maintenance is used as the reference to be compared with during the next incremental translation. The advantage of having a maintenance phase is that during the next iteration, the comparison will find less irrelevant changes and consequently need less user interaction.

Typical examples of changes made during the maintenance phase in SDL are: deleting an unnecessary management process; deleting unwanted newtypes generated from types used in variables or parameters; renaming a process or block instances that was given a default name; and

deleting channels to group communication routes. These kind of changes do not have to be translated back to UML and can be done during the maintenance phase.

V 10.6 Change Report

The last feature we propose to improve the round-trip engineering is the creation of a change report. During each iteration all the changes found in the model or specification are logged in a change report together with the scope and translation rules that were applied. The resulting report is a hypertext document with summary report with links to the model for the scope and a link to a description of the translation rule. When viewing a translation rule, the variables used in the rule are assigned concrete values. In a version control system, this document is stored together with the new version of the model or specification.

The change report is useful for several purposes. I can be used to find erroneous translation of changes. It can be used to document the changes or simply as a means to track changes in different versions. It can be used to navigate through the specification to find the spots that may need further development. Furthermore, this feature could be combined with the *interactive rule activation* technique. The change report is presented and the developer can choose to delete certain changes or rules in order to avoid wrong translations before they are applied.

VI. CONCLUSIONS

"It is almost impossible to watch a sunset and not dream."

-Bern Williams-

"Instead of thinking about where you are, think about where you want to be. It takes twenty years of hard work to become an overnight success."

-Diana Rankin-

VI. 1 Main Contributions

The research presented in this dissertation is an important contribution in bringing UML and SDL closer toward each other. We provide a complete mapping of UML class diagrams and state diagrams onto an SDL specification. Mismatches in the languages are solved by providing extra information (e.g. stereotypes) or by preprocessing the model (e.g. flattening nested state machines). Moreover, we developed a method to support round-trip engineering between two different paradigms that do not have a one-to-one mapping. During an iteration, only changes are translated and not the complete model. As long as an entity is not changed, the linked entities are completely untouched. This method maximizes the preservation of detailed design done during previous iterations. We applied this approach to allow round-trip engineering between UML and SDL.

We already started to put our experiences in the real world. The current mapping and translation rules are validated during a joint research effort with Telelogic and a summary was published in [VE99]. The core of the UML to SDL translator of Telelogic Tau 3.6 was developed based on these results. There were concrete plans to implement the round-trip support as presented in this dissertation, but because of time constraints and the size of the project, this has been postponed.

The idea of synchronizing two models on different abstraction levels by incrementally translating changes is a contribution on its own. By replacing the UML and/or SDL information models and rewriting the translation rules, the idea can be applied to other source and target languages.

The idea of synchronizing two models on different abstraction levels by incrementally translating changes is a contribution on its own and is applicable to other source and/or target languages. The only requirement is that there exists a (partial) translation between the two languages. To apply our approach, an internal information model must be created for both languages and the translation must be broken up into sets of translation rules for each individual entity. Additionally, the translation rules for deleting and modifying entities must be worked out.

Many of the criteria mentioned in the beginning of the dissertation (I 2.6) for the perfect UML-SDL round-trip solution have been accomplished. The forward incremental translation rules are defined in such a way that they can translate a complete UML model into an SDL specification. The translation rules are also designed to preserve as much information as possible. The graphical layout information contained in UML or SDL will be conserved as long as the structure or diagram does not have to be regenerated. Moreover, the generated SDL is made as readable as possible such that a developer can easily adapt the specification to his own needs. Finally, thanks to the possibility of re-linking entities, the round-trip support can still be provided after many iterations and when the system design and detailed design have evolved considerably.

On the other hand, the proposed UML-SDL solution is not perfect, as some of the criteria are not fulfilled. The reverse incremental translation rules can only be applied if there has been a forward translation first, so our solution cannot be applied to legacy SDL systems. We do not provide support to translate sequence diagrams into MSC's. There is no static check to ensure that the UML model and SDL specification are synchronized. Over time, they might diverge from each

other, e.g. when the user asks to ignore deleting some entities. Then again, allowing UML and SDL to evolve separately and still provide round-trip support can be considered a positive feature.

The goal of our research is to improve the development of large, high interactive systems. In other words, the systems' behavior should be suitable to be expressed in state machines. The system should benefit from the formal specification and validation features of SDL. In addition, the development of the system should require a high-level viewpoint to keep an overview of the system. It is clear that the UML-SDL round-trip engineering does not come completely free. First, developers have to invest in learning both UML and SDL. Given a development team with a lot of SDL experience and no UML experience, chances are high that adding UML in the process does not improve quality or time-to-market. Second, to guarantee a correct operation of the incremental translations in the round-trip process, the developers need to have insight in the UML-SDL mapping and adapt the system design and detail design accordingly. In other words, the round-trip engineering should be integrated in a full development process to bring its full benefit.

VI. 2 Future Work

Integrating the presented round-trip engineering in an existing UML and SDL tool requires more than implementing the translation rules one by one. The tool must be able to load a full SDL specification with al its peculiarities and graphical information, build the information model and write it back without loss of any details after it has been updated. This is a challenge because the presented information model does not cover the full SDL language. The same requirement holds for parsing, storing and writing back the UML model, including the diagram information. Furthermore, the translation rule definitions do not take error handling into account. Throughout all translation rules, extra checks must be performed to detect missing links or entities and to avoid duplication of information.

More research must be performed on the reverse engineering of a complete SDL specification. Our methodology requires a UML model to start the first iteration. This restriction makes it very hard to start using UML for continuing the development of an existing specification. Such reverse engineering support could perform an analysis of communication through channels and signal routes to discover the associations between classes and to find the correct scope for method definitions. An important issue in reverse engineering is the creation of class diagrams. As there is no one best way to create diagrams, the user must be given the option to choose from different type of diagrams, e.g. generalization tree, aggregation tree, diagram per class, etc. The results from this reverse engineering research can then be reused by extending the reverse translation rules and thus improving the incremental reverse translation. Especially the reverse engineering

The set of incremental translation rules that we propose in this thesis is not the only possible set of translation rules, nor is it necessarily the best possible set. Especially the translation rules that compare attributes of entities are subject for discussion. For example, an alternative for translating the change of the stereotype of a class from «block» to «process» is to regenerate the complete class, instead of reusing the management process linked to the class. Actually, there is no one best way to translate a particular change, as this probably depends on what the user intended with the change. In our translation rules, we choose the version that retains the most of the detail design decisions.

Currently, no support is provided to simultaneously change the UML model and the SDL specification. This poses practical problems if an analysis team and a design team work together on the same system. So, allowing simultaneous development in UML and SDL may be an important feature for the development of bigger systems. One way to realize this kind of support is to perform the forward iteration and immediately perform the reverse iteration without taking the updated SDL (i.e. result of forward iteration) into account. This scenario is likely to cause more conflicts and making the UML model and SDL specification less consistent. For example, if the name of an entity is changed on both sides simultaneously, there is no way to automatically decide which name is the correct one. More user interaction will be necessary to support this kind of synchronization. More research is needed to estimate the consequences and find appropriate solutions.

VI. 3 Related Research

This section describes existing research that is comparable to our round-trip process or that may be combined with it to improve it. In the current tendency, UML and SDL keep growing towards each other. This confirms the strong demand for combining and integrating the qualities of UML and SDL. The latest result in this area is the new version of SDL, namely SDL-2000. Because of the importance for our research, we discuss the impact of SDL-2000 to the UML-SDL round-trip engineering in more detail.

VI 3.1 SDL-2000

SDL-2000 is a major revision of SDL'96 and affects the UML-SDL round-trip engineering in many ways. In SDL-2000, outdated concepts are removed, existing concepts are aligned and new concepts are introduced. Many changes clearly improve the alignment with UML. For example, SDL-2000 now includes nested state diagrams, processes and blocks are unified in one concept called agents and non-delaying channels take over the role of signal routes. Moreover, SDL 2000 is accompanied by a new standard Z.109 SDL-UML profile. In general, the new concepts in SDL-2000 make SDL more suitable to combine with UML and allow improvements to the round-trip engineering process and translation rules. Below we describe the concepts that have an impact on the UML-SDL relationship.

In SDL-2000, blocks and processes are unified in the agent concept. Agents model the active components of a system; they have a communicating state machine with its own life cycle and signal input queue. SDL-2000 still differentiates between block agents and process agents, but the previous limitation that blocks and processes cannot be located in the same scope is eliminated. Moreover, variables can be declared within a block agent and are visible for the nested agents. If applied to our round-trip solution, a number of simplifications can be applied. The extra processes-block can be skipped, changing the stereotype of a class has fewer implications and «block» classes with attributes do not need the extra management process.

The alignment of the communication concepts in SDL-2000 makes the generated code easier to maintain. The signal route concept is purged and replaced by channels. A channel can be declared with the signals and without connections. The channel type can then be reused in different places throughout the system to connect block agents and process agents. Applied to our round-trip engineering, the UML association maps bi-directionally on an SDL channel type. The channel instances are only mapped in the forward direction. This eases the synchronization and the maintenance of the UML-SDL links.

SDL-2000's composite states allow hierarchical state machines. Just like in UML, states can be nested, agents can be in more than one state at a time and states can define an entry and exit procedure. This feature makes the flattening of the UML state diagram before translation unnecessary and provides a clearer link between UML states and SDL states.

In SDL-2000, UML-like class symbols can be used to refer to type definitions and diagrams. Multiple references are allowed, but they all have to be consistent with the type definition.

Relations between two classes can be depicted by associations and specialization. Associations do not imply any predefined semantics to the referred SDL agents. Specializations must be consistent with the *inherits*-clause in the referred type definitions.

Z.100 (SDL with UML) specifies how the UML notation may be used within an SDL specification and the Z.109 (UML to SDL) specifies how a UML model is translated into an SDL model. The purpose of the UML for SDL is to define a set of extensions and restrictions to UML to ensure an unambiguous mapping between the two languages. Based on the rules defined by Z.109 the UML to SDL tool translates UML static structure diagrams (class diagrams) together with state charts into SDL architecture and behavior. UML diagrams grouped into a package will automatically be transformed into an SDL packages or an SDL System.

The UML to SDL tool adds SDL semantics to the UML model. As a result, the abstract UML model becomes a formal specification, which is possible not only to compile but also to simulate. The benefit is that an architecture specification with unambiguous interfaces can be achieved. It is also possible to verify the dynamic properties of the system's interfaces. The resulting SDL system can then be used as the basis for further implementation in SDL and automatic code generation for the application.

To conclude, SDL-2000 is without doubt the next step toward the integration of UML and SDL. It would be an interesting challenge to make the best combination of our round-trip engineering solution with the updated language and mapping. Some of the tricky mapping problems are solved by the new concepts in SDL-2000 and the translation rules for changes can contain more semantics.

VI 3.2 UML for Real-Time

An interesting research track is to extend object oriented modeling languages with real-time aspects, such that source code can be generated directly from the high-level design model. The Real-Time Object-Oriented Modeling language (ROOM [SGW94]) specifically tailors object-oriented concepts for real-time systems. It offers a single consistent set of graphical modeling concepts with the benefits of object paradigm and executable models. After UML has turned into the defacto OOA language, UML and ROOM have been combined into UML for Real-Time (UML-RT, [Lyo98]) to deliver a complete solution for modeling complex real-time systems. In UML-RT, structure is described in terms of *capsules* and is modeled by class diagrams and collaboration diagrams. Behavior is described in terms of extended, hierarchical, finite state machines. Communication between capsules is based on synchronous or asynchronous message passing.

UML-RT introduces the concepts (i.e. stereotypes) capsule, port and connector to support the modeling of real-time systems. Capsules correspond to the ROOM concept of actors. Capsules are complex, potentially concurrent, and possibly distributed active architectural components. They interact with their surroundings through one or more signal-based boundary objects called ports. Collaboration diagrams are used to describe the structural decomposition of a Capsule class. A port is a physical part of the implementation of a capsule that mediates the interaction of the capsule with the outside world. Ports realize protocols, which define the valid flow of information (signals) between connected ports of capsules. By forcing capsules to communicate solely through ports, it is possible to fully de-couple their internal implementations from any direct knowledge they have about the environment. This de-coupling makes capsules highly reusable. Connectors capture the key communication relationships between capsules. These relationships have

architectural significance since they identify which capsules can affect each other through direct communication.

The functionality of simple capsules is realized directly by the state machine associated with the capsule. Capsules that are more complex combine the state machine with an internal network of collaborating sub-capsules joined by connectors. These sub-capsules are capsules in their own right, and can themselves be decomposed into sub-capsules. This type of decomposition can be carried to whatever depth is necessary, allowing modeling of arbitrarily complex structures with just this basic set of structural modeling constructs. The state machine (which is optional for composite capsules), the sub-capsules, and their connections network represent parts of the implementation of the capsule, and are hidden from external observers.

Current UML-RT tools (ObjectTime Developer and Rational Rose Real-Time) provide model execution capabilities, and automatically generate complete code for complex real-time applications from these modeling constructs. Unfortunately, UML and UML-RT still lacks a precise dynamic semantics necessary for formal specification.

Of special interest in our context is that originators of UML-RT present a mapping of SDL to UML-RT in [SR99]. In this paper, they describe a transformation of an SDL system specification into a UML-RT model. The purpose behind such a translation is to take advantage of the formalized system specification of SDL and the versatility of UML with its broad acceptance and tool support. The mapping is describes in informal text. The SDL structural concepts map on capsules. Channels and signal route endpoints as well as gates map on ports. The outgoing and incoming signal lists of the gate serve to define protocol definitions. The channels and signal routes map directly to UML-RT connectors. As compared to our UML-SDL mapping, they provide some extra mappings such as the SDL save signals, create statement, decisions and timers.

Our conclusion is that the mapping of SDL onto UML-RT is closer to a one-on-one mapping than the mapping we propose. However, this does not necessarily mean that it is more useful in all circumstances. The mapping on UML-RT is better to examine the exact structure of the SDL specification. Our mapping on UML is better to give a view on a higher abstraction level and to document different views on the system.

VI 3.3 Version Management

Our round-trip-engineering process can profit from the integration with a version management system. In our current approach, a copy of the model and specification is stored after each iteration. This copy is used as the “old” model or specification in the next iteration. This approach can be improved by integrating a version management system. Because such a system keeps track of all intermediate versions, it is possible to backtrack to an earlier version and to accumulate changes and their translations in two or more versions. If the subsequent versions are stored in the form of delta’s, the version management can take over part of the comparison process. Moreover, a version management system is almost indispensable for larger systems, where the UML model and the SDL specification are managed by different teams.

VI 3.4 Round-Trip Engineering Solutions

Many round-trip engineering approaches exist between UML and most object-oriented programming languages. The one-on-one mappings between UML concepts and their Java or C++ counterparts allow direct synchronization of the model with the source code. For example, if a

Java class has variable *x*, but the corresponding UML class does not have an attribute *x*, then the attribute is added to the UML class.

The previous generation of round-trip engineering tools polluted the code with comments generation tags to indicate the part of the code that may be modified. Newer tools provide the same functionality without the need of these tags. In most UML tools with support for round-trip engineering (e.g. Rational Rose, Paradigm Plus and Rhapsody) the developer executes a incremental forward translation after changing the model or an incremental reverse translation after changing the code to synchronize the model with the code. Together/J gives a more advanced support for round-trip engineering in the sense that the UML models are *always* synchronized to the source code that implements them. Change something in a Class or other source generating diagram and the relevant source code updates immediately. Change the code and the visual model updates to stay in sync.

These tools provide almost the perfect solutions according to the criteria given in section I 2.6. The iteration can be started by reverse engineering the source code; most information of the static structure is translated; the model and the source code is guaranteed to be synchronous after an iteration; the iteration can be applied over and over again and in recent versions, the source code is not polluted with round-trip specific comments. Criteria that these solutions do not provide are: the UML state diagram is not translated and the design model and implementation model cannot diverge, even if the developer would like to.

Unfortunately, the UML based round-trip solutions mentioned above are very hard to apply to SDL as a programming language because of the complex mapping. The new features of SDL 2000 and its relation with UML might improve the possibility to use existing approaches of round-trip engineering. SDL 2000 provides a limited set of one-on-one mappings between UML and SDL. These mappings could be used for direct synchronization of a UML model and an SDL specification. However, further research is necessary to find out to which extend this is possible and whether the resulting round-trip solution generates enough SDL code. One of the tedious parts of writing SDL –creating gates and communication routes- would probably be hard or impossible to support by such round-trip engineering tool.

REFERENCES

> Every man I meet is in some way my superior.
>
> - Ralph Waldo Emerson-

[AC99] *Andersen Consulting Foundation Software Organization.* http://www.ac.com/services/foundation/foun_home.html

[AIA98] Advanced Internet Access project description, ITA-2 AIA consortium, Brussel, November 1998.

[And95] E. Anders. *The SOMT Method*. Preliminary product information, Telelogic, September 19, 1995.

[Bau96] C. Baudoin, G. Hollowell. *Realizing the Object-Oriented Lifecycle*. Upper Saddle River, NJ: Prentice Hall, 1996.

[Beck99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Boe76] B. Boehm. *Software Engineering*. IEEE Transactions on Computers, vol. G-25, no.12, December 1976, pp.1226-1241.

[Boe81] B. Boehm. *Software Engineering Economics.* Englewood Cliffs, Prentice-Hall.

[Bræ96] R. Bræk, et al. *SISU Integrated Methodology - at a glance*, Oct 96.

[BR95] G. Booch, J. Rumbaugh. *Unified Method for Object-Oriented Development.* Rational Software Corporation, 1995.

[BH93] R. Braek, Ø. Haugen. *Engineering Real Time Systems*, Prentice-Hall, 1993.

[CAB94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes. *Object-Oriented Development, The Fusion Method.* Prentice Hall International Editions, 1994.

[Cin] Cinderella. http://www.cinderella.dk

[Dou98] B. Douglass. *Real-Time UML, Developing Efficient Objects for Embedded Systems.* Addison-Wesley, Massachusetts, 1998.

[DW98] D. D'Souza, C. Wills. *Objects, Components and Frameworks with UML : The Catalysis Approach*. Addison-Wesley, 1998.

[EHS97] J. Ellsberger, D. Hogrefe, A. Sarma. *SDL, Formal Object-Oriented Language for Communicating Systems*. Prentice Hall, London, 1997.

[GHY97] I. Graham, B. Henderson-Sellers, and H. Younessi. *The OPEN Process Specification*. Addison-Wesley, 1997.

[Har87] D. Harel. *Statecharts: a Visual Formalism for Complex Systems.* Science of Computer Programming 8, 1987, 231-274.

[Har97] David Harel and Eran Gery. *Executable Object Modeling with Statecharts,* IEEE Computer Magazine, July 1997, pp 31-42.

[Hig00] J. Highsmith. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House, 2000.

[HWW96] E. Holz, M. Wasowski, D. Witaszek, S. Lau, J. Fischer, P. Roques, K. Verschaeve, E. Mariatos, and J.-P. Delpiroux. *The INSYDE Methodology*. Deliverable INSYDE/WP1/HUB/400/v2, ESPRIT Ref: P8641, January 1996.

[IEEE83] Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology.* New York, NY: 1983.

[INS94] INSYDE. *Technical Annex: "Integrated Methods for Evolving System Design"*, ESPRIT-III Project P8641, restricted report edition, December 1994.

[ITA98] ITA-2 - Information Technology Access program, second part. http://www.iwt.be

[ITU94] ITU-T. *Z.100, CCITT Specification and Description Language (SDL)*, June 1994.

[ITU94-2] ITU-T. *Z.120, Message Sequence Chart.* Sep. 1994.

[ITU99] ITU-T. *Recommendation Z.109, SDL combined with UML (SDL/UML).* 1999

[ITU00] ITU-T. *Methodology on the use of ITU description techniques: SDL, MSC, ASN.1, ODL and TTCN*. Supplement 1 to ITU-T recommendation z.100.

[Jac94] I. Jacobson. *Object-Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, 1994.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *Unified Software Development Process*. Addison-Wesley, 1999.

[Kru99] Philippe Kruchten. *Rational Unified Process-An Introduction*. Addison-Wesley, 1999.

[LKH99] F. Lodge, K. Kimbler, M. Hubert. *Alignment of the TOSCA and SCREEN Approaches to Service Creation.* IS&N'99, LNCS 1597, pp. 277-290, 1999.

[Lyo98] A. Lyons. *UML for Real-Time Overview*. ObjecTime, April 1998.

[MCD99] Paul Monday, James Carey, Mary Dangler. *SanFrancisco Component Framework: An Introduction*. Addison-Wesley, 1999.

[MS97] G. Melby, R. Sanders (ed). *Sluttrapport for SISU II*, Mar 97.

[Mel99] Steve Mellor. *Automatic Code Generation from UML Models*. C++ Report, June 1999.

[Nav93] Z. Navabi. VHDL Analysis and Modeling of Digital Systems. McGraw-Hill, Inc., 1993.

[OMG99] OMG (Object Management Group). *Unified Modelling Language (UML) v. 1.3 specification*, http://www.omg.org, June 1999.

[PJH95] J. Peeters, M. Jadoul, E. Holz, M. Wasowski, D. Witaszek, and J.P. Delpiroux. *Hw/sw co-design and the simulation of a multimedia application*. In 7th European Simulation Symposium, October 1995.

[Rat97] Rational Software Corporation, et al. *UML 1.1 Semantics.*

[Rat99] Rational Unified Process 5.5, Rational Software Corporation, http://www.rational.com.

[Rat00] Rational Software Corporation. *Rational Rose 2000.* http://www.rational.com.

[RBP91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RGG96] E. Rudolph, J. Grabowski and P. Graubmann. *Tutorial on Message Sequence Charts (MSC'96)*. Tutorial of the Forte/PSTV'96 conference, Germany (October 1996).

[Roy70] W. Royce. *Managing the Development of Large Software Systems: Concepts and Techniques.* Proceedings WESCON, Aug. 1970.

[SCR98] SCREEN deliverable D28, *SCREEN Engineering Practices for Component-based Service Creation*, SCREEN/A21-D29, ACTS SCREEN consortium, December 1998.

[SGW94] B. Selic, G. Gullekson, and P. Ward. Real-Time Object-Oriented Modelling. IEEE Computer Society, 1994.

[SCVM95] D. Sinclair, L. Cuypers, K. Verschaeve, V. Mariatos, N. Kyrlogou, J.L.Roux (1995). *A formal approach to HW/SW Co-design: The INSYDE Project*. 9th International IEEE Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'96), Friedrichshafen, Germany.

[Sta97] J. Stapleton. *Dsdm Dynamic Systems Development Method : The Method in Practice*. Addison Wesley, 1997.

[SPC94] Software Productivity Consortium. *Using New Technologies: A Technology Transfer Guidebook*, 1994.

[SR99] B. Selic, J. Rumbaugh. *Mapping SDL to UML*. Rational Software white paper, 1999.

[Tele] Telelogic, www.telelogic.com

[TOA95] *A Comparison of Object-Oriented Development Methodologies.* The Object Agency, Inc., 1995.

[Tog00] TogetherSoft. *Together Enterprise*. http://www.togethersoft.com.

[TOS98] TOSCA deliverable D9, *Specification of the TOSCA Process Architecture for Service Creation*, ACTS TOSCA consortium, December 1998.

[Ver97] K. Verschaeve. *Automated Iteration between OMT* and SDL.* Proceedings of the Eighth SDL Forum (SDL'97), Evry, France, September 1997.

[VE99] K. Verschaeve, A. Ek. *Three Scenarios for Combining UML and SDL'96.* Proceedings of the Ninth SDL Forum (SDL'99), Montréal, Québec, Canada, June 1999.

[VJW96] K. Verschaeve, V. Jonckers, B. Wydaeghe, L. Cuypers. *Translating OMT* to SDL, Coupling Object-Oriented Analysis with Formal Description Techniques*. Proceedings of Method Engineering '96, p.126-141. IFIP, Atlanta, USA, 1996.

[VWCJ95] K.Verschaeve, B. Wydaeghe, L.Cuypers, V. Jonckers, J. Heirbaut. *OMT*, Bridging the Gap between Analysis and Design*. Proceedings of 8th International Conference on Formal Description Techniques (FORTE'95), Montreal, Canada, 1995.

[VWW00] K. Verschaeve, B. Wydaeghe, F. Westerhuis, J. De Moerloose (2000). *Multi-level Component Oriented Methodology for Service Creation*. Proceedings of 7th International Conference on Intelligence in Services and Networks (IS&N 2000), Athens, Greece.

[VWW01] K. Verschaeve, B. Wydaeghe, F. Westerhuis, J. De Moerloose (2000). *Visual Composition with SDL Beans*. Submitted to International Conference on Engineering of Computer Based Systems (ICBS 2001), Athens, Greece.

[WVMV98] B. Wydaeghe, K. Verschaeve, B. Michiels, B. Van Damme, E. Arckens, V. Jonckers (1998). *Building an OMT-Editor Using Design Patterns: An Experience Report*. Tools '98: The Move to Componentware. St. Barbara, 1998

[WWV95] M. Wasowski, D. Witaszek, K. Verschaeve, B. Wydaeghe, E. Holz, and V. Jonckers. *The complete OMT**. Deliverable INSYDE/WP1/HUB/300/v3, ESPRIT Ref: P8641, December 1995.

[VWWM00] Multi-level Component Oriented Methodology for Service Creation, K. Verschaeve, B. Wydaeghe, F. Westerhuis, and J. De Moerloose, IS&N 2000, Springer, Athens, February 2000, ISBN 3-540-67152-8.