Reverse Engineering SDL Models. A Pattern-Based Approach

Evert Arckens
System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, 1050 Brussel, Belgium
+32 2 629 29 88 (tel), +32 2 629 28 70 (fax)
earckens@info.vub.ac.be

Abstract

SDL models are often intentionally or unintentionally developed using certain patterns. When reverse engineering SDL models, it is interesting to extract the pattern information from the SDL models. This paper describes translation rules and guidelines for the translation of SDL models to OMT models, and it describes the detection of SDL patterns and their notation in UML.

Keywords
SDL, OMT, UML, Reverse Engineering, Patterns

1 INTRODUCTION

Many companies are using systems that have been developed several years ago. During all those years, these systems have been adapted for maintenance or changed requirements. In many cases this happened without maintaining documents, or without keeping existing documents consistent. Many of those systems are now difficult or even impossible to understand and maintain.

The above scenario is also true for SDL systems. Many SDL systems that exist today have been developed without using any kind of analysis and design methodology. However today, designers are gaining interest in using object oriented analysis and design methodologies like the INSYDE methodology [1].

The INSYDE methodology consists of different stages. The first stage is the analysis, for which the OMT formalism is used. This stage is followed by the system design, for which OMT* is used. OMT* is a formal subset of OMT, based on the syntax of OMT and on the semantics of SDL. In a third stage, the OMT* models are translated into SDL models. For this translation, translation rules and guidelines have been developed [2]. The fourth stage is detailed design. The SDL models are extended into full SDL models that can be simulated or translated into source code. The fifth and last stage is the validation stage. In the INSYDE methodology the SDL models are validated by simulating the system.

To make the introduction of a methodology like the INSYDE methodology easier, it is best to integrate the existing SDL models with the methodology instead of ignoring the existing systems and design them all over again. To integrate the existing SDL models, documentation and analysis models have to be extracted, or to be 'reverse engineered', from the SDL models. In our research these analysis models are represented as OMT or UML models. This documentation can then be used to understand, maintain or re-engineer the systems.

In section 2 we will discus some translation rules and guidelines we have created to reverse engineer SDL models into OMT* models. In section 3 we will then discuss how we can combine the reverse engineering of SDL models with SDL patterns.

2 SDL TO OMT*

As stated in the previous section, the INSYDE methodology provides translation rules and guidelines for the

translation of OMT* models into SDL models, and to extend those translated SDL models into 'full' SDL models. As a first step we have reversed those translation rules and guidelines. We then adapted these with the aim to provide translation rules and guidelines that translate as much as possible of the SDL models into OMT* models.

This resulted in a set of translation rules which provide a mapping between SDL elements and OMT* elements. In the next section we will discus this mapping and some of the translation rules.

2.1 Translation rules

In 'Table 1' an overview is given of the mapping between SDL elements and OMT* elements. These elements can be mapped onto each other by applying the translation rules. We will explain a few of them.

SDL	OMT*
Specification	Model
Package, System	Module
Environment	Class 'ENV'
Package reference	'dummy' class
Block type, Block	Class
Typebased block	Aggregation
Process type	/
Process, Typebased process	Class, State diagram
Signal declaration	/
Variables	Attributes
Channel & signal route	Association
State diagram	State diagram
State	State
Start state	Initial state
Stop state	Final state
Input signal	Event & operation
Task	Action
Output signal	Output event
Decision	>1 guarded transitions

Table 1 : Mapping between SDL and OMT*

The translation rules are built in such way that they call each other for the translation of a particular element. The translation of a complete SDL specification is started with the translation rule for a specification, which will call the translation rules for the packages and system.

System and environment

An SDL system is translated into an OMT module. While creating this module, a passive class with the name "ENV" is put into this module to represent the environment of the system.

Typebased block

A typebased block is contained in another structure element and is of a certain block type. The structure and the block type both map onto a class. The typebased block is then translated by putting an aggregation between those two classes, where the structure class is the aggregate, and the block type is the component. On the component side we put the name of the typebased block as a role.

Process, process type and typebased process

If a block contains only one process, then only the state diagram of the process is translated into the state diagram of the class that corresponds to the block. If the block contains more than one process, then a class is created for each process, each containing the translated state diagram of the respective process.

A process type is not translated. It's information is only used for translating a typebased process. A typebased process is translated as if it is a normal process, but with the information of the process type.

In 'Figure 1' the translation is shown of a block containing two processes and a typebased process.

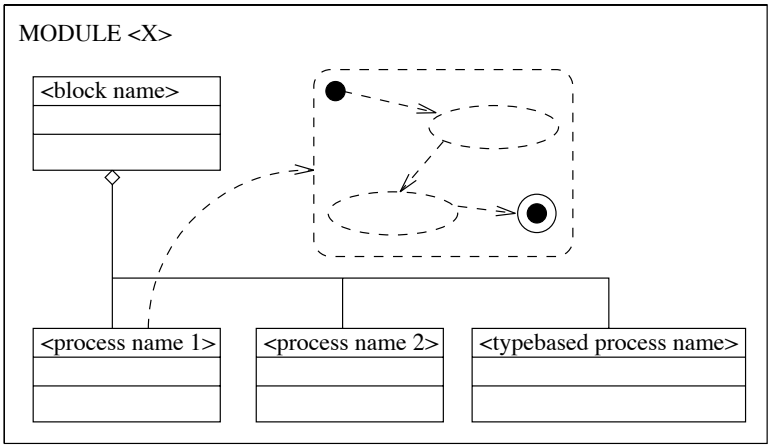

Figure 1: Multiple processes

State diagram

As stated above, for each process their exists a class in the OMT* model. Either a new class or the class of the block to which the process belongs. The state diagram of the process is then translated into a state diagram of the corresponding class. SDL and OMT* state diagrams are very similar, and can thus be mapped onto each other fairly easy, as shown in 'Table 1'.

Input signal

An input signal is mapped onto two OMT* elements instead of one.

In 'Table 1' it is shown that a signal declaration in itself is not translated. The translation of a signal is delayed until the place where a signal is used as an input signal is reached. The signal is then translated into an operation of the corresponding class.

Simultaneously, the input signal is translated to an event on a transition between two states in the corresponding state diagram.

Decision

The concept of a decision does not exist in OMT*. A very similar concept however is the guarded transition. A decision can be translated into a set of guarded transitions, by converting each question and answer couple into a boolean expression like: [question==answer]. For the 'else' part of the decision, the negation of all the other guards is taken. This is shown in 'Figure 2'.

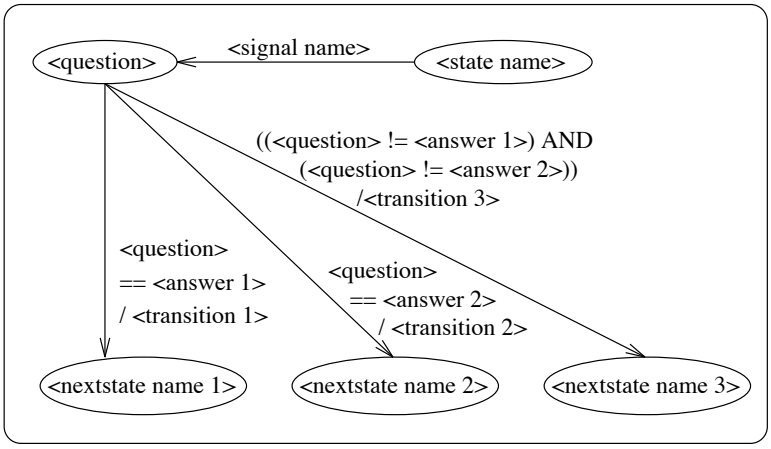

Figure 2: Decision as guarded transitions

'Figure 3' shows a graphical overview of the mapping between SDL and OMT*.

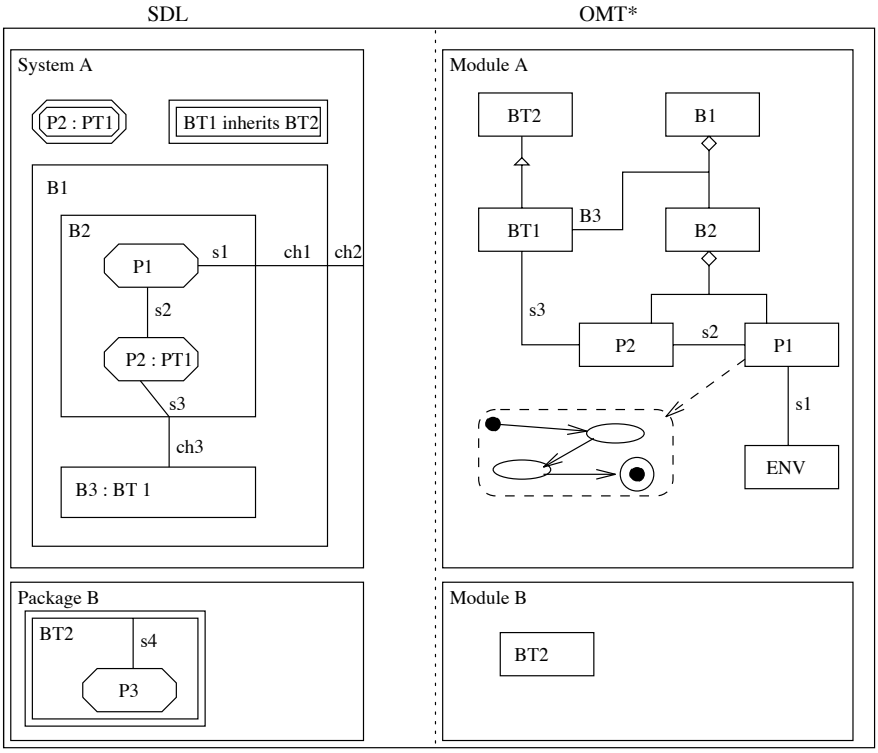

Figure 3: Graphical overview

2.2 Pre-processing

Not every SDL element can be mapped onto an OMT* element. To be able to perform an automatic translation, the SDL models have to be 'pre-processed', so that they only contain translatable elements. Some examples of these non-translatable elements are: Timer, Priority input, Addressing, Create request, Saving signals, and some shorthands like: Signal list, All signals, State list, All states, Same nextstate and Implicit signal routes and channels.

Before translating the model, these non-translatable have to be removed, or the information that they represent has to be described in another way. Many of these elements, like the shorthands, can be handled automatically. Others have to be changed by the user.

A timer can for instance be changed into informal text, which can just be copied into the OMT* state diagram.

Signal routes and channels are translated into associations. Implicit signal routes and channels can however not be guessed by a tool. They therefor have to be made explicit by a user.

2.3 Case studies

Based on the above described translation rules and guidelines, we've build a prototype of an SDL to OMT* translator. This prototype is written in Java. It can read SDL '.pr' files, and produces '.cd' and '.sd' files that can be read by the commercial OMT tools, or our own OMT editor [3].

We've tested this prototype on some case studies, among which the 'Toffee vendor' described in 'SDL Formal Object-Oriented Language for Communication Systems' [4], an 'Access control' (developed by Therese Nilsen of SINTEF Telecom and Informatics Norway), and a description of a 'TCP layer' protocol [5].

Toffee Vendor

The 'Toffee vendor' is a relatively small model. The translation of it results in an OMT* model that is very similar to the original SDL model. This is mainly due to the fact that it was very cleanly developed. Only a small amount of pre-processing has to be performed. Most of this pre-processing can be handled automatically, except for the removal and transformation of some addressing and a timer.

Access Control

The 'Access Control' system is a mid-sized system that describes the electronic access control to a door. In

this system, special attention has been paid to the object-oriented style of the system. After translation of this system, it can be seen that also the inheritance relationships that are in the SDL model can be found in the OMT* model.

TCP-Protocol
The SDL model of the TCP-Protocol mainly consists of informal text in the form of pseudo code. In this case the translator isn't very useful. The model consists of one block and two processes that can be translated. The pseudo code can only be copied as documentation. It is not further helpful in the translation.

2.4 UML

In the previous sections we've described some translation rules and guidelines for the reverse engineering of SDL models to OMT* models. We are now converting these rules and guidelines to use UML instead of OMT. First of all, UML is becoming a 'de facto' standard in industry. Secondly, UML is much 'richer' than OMT. This offers far more possibilities for the reverse engineering of SDL. In UML, different views of the same model can be generated, and also other kinds of diagrams, like collaboration diagrams, can be used. Other useful elements in UML that were not available in OMT are signal events, packages, stereotypes and patterns.

In the next section we will already make use of UML instead of OMT. Especially the use of patterns in UML in combination with the reverse engineering of SDL models will be discussed more elaborately.

3 PATTERNS

3.1 Reverse engineering and SDL patterns

The translation rules we discussed in section 2 translate SDL models element by element. To get a good overview of the system, it is important to find the more global structures and interactions of the SDL system, and to document these in the UML models. In other words, we have to move to a larger granularity. This brings us to the research area of patterns and design patterns.

A lot of work has been done in recent years in the area of patterns. There exist all sorts of patterns [6], like object-oriented analysis patterns [7], object-oriented design patterns [8], domain specific patterns, and so on. Object-oriented analysis patterns and object-oriented design patterns concentrate on the structuring of object-oriented analysis and design models. The domain specific patterns are used to describe domain knowledge. They describe structures and interactions that are typical for a specific domain. These are the kind of patterns we are interested in. In 'Pattern-based Configuring of a Customised Resource Reservation Protocol with SDL' [9] and 'Configuring Communication Protocols Using SDL Patterns' [10], a definition of SDL patterns and some examples of SDL patterns that are typical for the domain of telecommunication applications are given.

The reverse engineering of SDL models consists of three parts. First the models have to be translated element by element to UML models as described before. Then the patterns have to be detected, and finally they have to be represented in the UML models. This detection en representation will be described in the next sections.

3.2 Detecting patterns

Patterns can not only be detected in systems that have been developed with the patterns in mind. They can also be detected in other systems. This is due to the fact that new patterns are developed when their principle or their structure has proven useful in a lot of systems already.

To detect patterns in SDL models, we must first describe what makes a pattern detectable. In 'Design Reverse-Engineering and automated design pattern detection in Smalltalk' [11], Kyle Brown states that: 'In general, a pattern is detectable if its template solution is both distinctive and unambiguous.' This statement was made for design patterns in Smalltalk programs, but it is true for the detection of any kind of patterns. With distinctive is meant that the template solution has a diagram that is only used when the pattern is present, and for nothing else. Unambiguous means that the template solution of a pattern can be described with only one diagram.

A pattern description consists of a set of elements, among which a semantical description and a syntactical template solution. It is this template solution that can be used as a starting point for the detection of patterns in SDL models. Since this is only a template, the real embedding of the pattern in the model can vary. This means that a pattern can not always be automatically found. We have to look for syntactical elements that give an indication of the presence of a certain pattern. For some patterns it will be possible to find them automatically, others will have to be found with the aid of a user.

After a first investigation of the six patterns that are described in [9,10] (BlockingRequestReply, Codex, TimerControlledRepeat, DuplicateIgnore, DuplicateHandle and DynamicEntitySet), we made the following

classification.

BlockingRequestReply and TimerControlledRepeat
The BlockingRequestReply and TimerControlledRepeat patterns can be detected fairly easy. The syntactical template of the BlockingRequestReply patterns is shown in figure ??. This pattern can be found by looking for a transition in a certain service or process (Automaton_A), where a message is sent to another service or process (Automaton_B), and after which nothing is done until Automaton_A receives a reply from Automaton_B. This is shown in the top half of figure ??. The state diagram of Automaton_B must be such that after receiving the message from Automaton_A, it will always eventually send a reply to Automaton_A. This is shown in the bottom half of figure ??. The only exception that is allowed on the fact that Automaton_A does nothing between sending a message and receiving the reply, is when this pattern is combined with the TimerControlledRepeat pattern. Right after sending the message, a timer can be set. Although the setting of the timer could be done just before sending the message, we have to take this possibility into account when searching for the pattern.

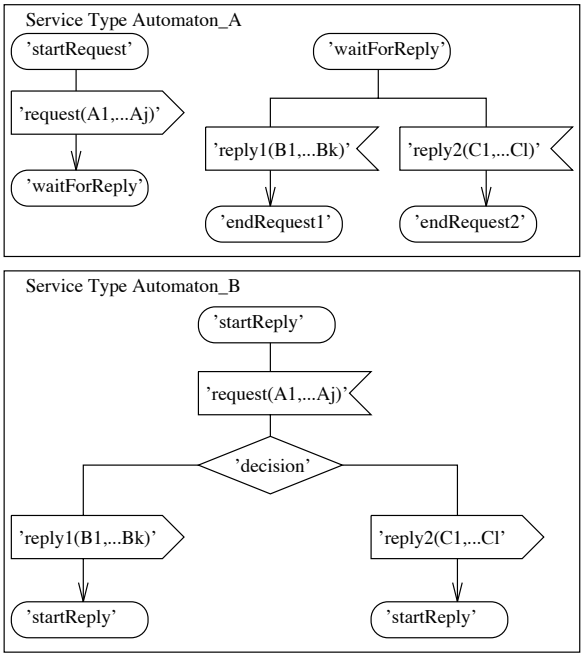

Figure 4: BlockingRequestReply

The TimerControlledRepeat pattern can be found by looking for two transitions that follow each other. One in which a message is sent and a timer is set, and another in which the same message is sent again when the timer signal is received.

DynamicEntitySet
When a process is found that creates instances of another process upon request, and then forwards messages to it, it could be an EntityAdministrator. So, a DynamicEntitySet pattern could be present. Although the template solution is not completely distinctive, it could be indicative. A designer can then be asked to confirm or deny the presence of the pattern.

Codex, DuplicateIgnore and DuplicateHandle
The template solutions of the Codex, DuplicateIgnore and DuplicateHandle patterns are small and can be filled in with a large amount of freedom. This has as a result that the patterns are not very distinctive, and can therefor not be easily detected automatically. A designer will have to point them out.

3.3 Patterns in UML models

After an element by element translation of SDL models and the detection of patterns in those SDL models, the pattern information has to be put into the newly created UML models. In this section we describe the notation that is available in UML for patterns. We then give an example of how this notation can be used for SDL patterns.

UML Notation

In the UML notation guide [12], the definition of a notation for patterns in UML models is given. In 'UML Toolkit' [13] and 'Applying UML and Patterns' [14] it is shown how this notation can be used for the description of design patterns. This notation was mainly developed for object-oriented design patterns, like those described in 'Design Patterns. Elements of Reusable Object-Oriented Software.' [8]. But after a first investigation, this notation seems perfectly applicable for the domain specific SDL patterns as well.

The notation of patterns in UML consists of two main parts. First, the participating classes are indicated in a static class diagram. The complete pattern is represented by one single symbol, a dashed ellipse, in which the name of the pattern is written. The ellipse is connected to the participating classes with dashed lines, on which the roles are written that these classes play in the context of the pattern (see figure ??). The second part consists of a collaboration diagram. In this diagram it is described how the classes work together in the context of the pattern. A collaboration diagram has to be seen as a parameterized diagram. The classes in the collaboration diagram are to be filled in by the classes that have been indicated in the static class diagram (see figure ??).

In the context of reverse engineering SDL models, it will be the first part that is created. When a certain pattern is detected, a dashed ellipse representing the pattern and connected to the participating classes is placed in the static class diagram. The second part, the collaboration diagram, can be seen as known information about the pattern. It is information that can be placed in a library, and to which can be referred when a certain pattern is found.

Example

As an example, we again refer to the BlockingRequestReply pattern.

Figure ?? shows the static class diagram of a system in which a BlockingRequestReply has been used. It is the result of reverse engineering an SDL model. The dotted parts represent the rest of the system. The interesting parts are the classes Automaton_A and Automaton_B. Automaton_A can accept the signals 'reply1' and 'reply2'. Automaton_B can accept the signal 'request'. The ellipse represents the BlockingRequestReply pattern. The dashed lines show that the class Automaton_A plays the role of 'requester' in the pattern, and that the class Automaton_B plays the role of replier.

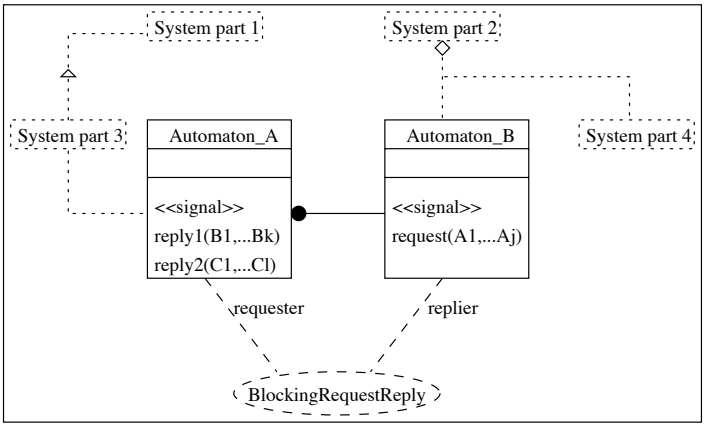

Figure 5: Static class diagram

Figure ?? shows the collaboration diagram that is associated with the BlockingRequestReply pattern. The objects 'Requester' and 'Replier' serve as parameters for the collaboration. They will be filled in by the respective classes that are indicated by the roles in the static class diagram. In our case this is respectively Automaton_A and Automaton_B.

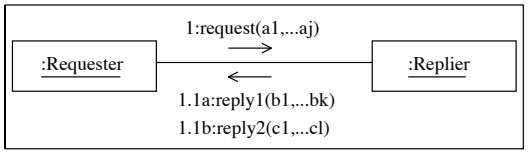

Figure 6: Collaboration diagram

Extensions

Although the existing notation for patterns in UML models is applicable, it would be interesting to see if there more possibilities to represent SDL patterns in UML models. Stereotypes could for instance be used to create

specific kinds of classes that play a certain role in a pattern. Although it would not be possible then for that class to participate in multiple patterns. Another example is the usage of state diagrams. In the definition of SDL patterns, a lot of state diagrams are used. It would be interesting to also use the state diagrams of the UML models to describe the patterns.

4 CONCLUSIONS AND FUTURE WORK

In this paper, we've presented a technique to reverse engineer SDL models. We've first presented some translation rules and guidelines to translated SDL models into OMT* models. These translation rules and guidelines can be easily ported to use UML instead of OMT*.

In the second part of this paper we've discussed the combination of reverse engineering SDL models and the use of SDL patterns. To extract information about the used SDL patterns is very valuable into understand, maintain and re-engineer SDL models.

The detection of patterns shows to be a hard task, for which a lot of research still has to be done. It could for instance be researched in what way a run-time (or simulation-time) analysis of the SDL models could help to detect SDL patterns. Another interesting topic would be to find a generic technique to detect SDL patterns. This would help to avoid to create a new detection method for each new SDL pattern that is created.

The existing notation for patterns in UML is perfectly applicable for our purpose. Although some extensions with stereotypes and state diagrams could be considered.

Finally, the scalability of our technique has to be tested.

5 ACKNOWLEDGEMENTS

This paper has been conducted by a grant of the IWT (www.iwt.be) number: 961080.

6 REFERENCES

[1] E. Holz; M. Wasowski; D. Witaszek; S. Lau; J. Fisher; L. Cuypers; J. Heirbaut; K. Verschaeve; V. Jonckers: "The INSYDE methodology report.", INSYDE/WP1/HUB/400/v3 Esprit Ref: P8641, 1994.

[2] V. Jonckers; K. Verschaeve; B. Wydaeghe; L. Cuypers; J. Heirbaut: "Bridging the gap between analysis and design.", Proc. of The 8th International Conference on Formal Description Techniques, FORTE'95, Montreal, Canada, 1995.

[3] B. Wydaeghe; K. Verschaeve; B. Michiels; B. Van Damme; E. Arckens; V. Jonckers: "Building an OMT-editor using design patterns: An experience report.", Accepted at TOOLS'98, 1998.

[4] J. Ellsberger; D. Hogrefe; A. Sarma: "SDL Formal Object-Oriented Language for Communicating Systems.", Prentice Hall, 1997.

[5] D. Prakash: "SDL specification of transport layer protocols.", Master's thesis, Vesalius College, Vrije Univesiteit Brussel, Brussels, Belgium, 1995.

[6] W. Pree: "Design Patterns for Object-Oriented Software Development.", Addison-Wesley, ACM Press, 1995.

[7] M. Fowler: "Analysis Patterns: Reusable Object Models.", Addison-Wesley, 1997.

[8] E. Gamma; R. Helm; R. Johnson; J. Vlissides: "Design Patterns. Elements of Reusable Object-Oriented Software." Addison-Wesley, 1995.

[9] B. Geppert; F. Grössler: "Pattern-based configuring of a customised resource reservation protocol with SDL." Technical report, SFB501 19/96, Computer Networks Group, University of Kaiserslautern, Germany, 1996.

[10] B. Geppert; R. Gotzhein; F. Grössler: "Configuring communication protocols using SDL patterns.", Proc. of The 8th SDL Forum, "SDL'97 Time for testing. SDL, MSC and Trends", Elsevier, 1997.

[11] K. Brown: "Design reverse-engineering and automated design pattern detection in Smalltalk.", http://www.ksccary.com/brown.htm.

[12] G. Booch; I. Jacobson; J. Rumbaugh: "The UML specification documents.", Rational Software Corp., Santa Clara, CA, 1997. See documents at http://www.rational.com

[13] H-E. Erikson; M. Penker: "UML Toolkit", Wiley Computer Publishing, 1998.

[14] C. Larman: "Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design.", Prentice-Hall, Inc., 1998.