Crossing the Borderline – From Formal to Semi-Formal Specifications

Andreas Bollin

Institute for Informatics-Systems
University of Klagenfurt, Austria
Andreas.Bollin@uni-klu.ac.at

Abstract. Being part of the systems' documentation formal specifications can play a crucial role in the software development process. However, besides dense mathematical expressions, their semantical compactness and lack in visually appealing notations impede their use and comprehensibility among different stakeholders. One solution to this problem is to enrich the specification by a semi-formal view, in most cases diagrams with a sufficiently understood semantic meaning. However, up to now existing approaches only cover statics-bearing diagrams as control- and data-dependencies within declarative specifications are hard to detect.
This paper presents an approach for dependency analysis within declarative formal specifications. These dependencies are easily deduced, and UML diagrams showing static and dynamic properties of the specification get feasible the first time.

1 Introduction

Formal software specifications are usually recommended as means to produce high-quality software. They can solve the verification problem ("do the system right"). But, even if the system has been refined correctly, there is another problem to be solved: the validation problem ("do the right system").

What sounds like a requirements elicitation problem also has to do with the problem of choosing a suitable notation. The risky part is that the stakeholders of the project (developers, customers, authorities) have to agree upon the meaning of the formal specification. Here, unclear requirements and specifications lead to futile validations easily. And this, in consequence, leads to a "buggy" system. So, the problem is not the formal notation, as its semantics is well-defined. The problem is the likely misinterpretation of concepts – due to the different habits of the stakeholders.

So why not just combining formal specifications and wide-spread semi-formal approaches? Such a combination has several advantages. Different views (either of graphical or textual/mathematical nature) convey the concepts much better between different stakeholders. The approaches are not meant to replace each other, but they extend the possibilities of concept description: properties of semi-formal descriptions get deducible (by stepping into the formal world) and formal specifications can be described at an even more abstract level.

Several research teams are working on the issue of mapping graphical approaches (e.g. UML) to formal specification languages. The generated specification is then used for consistency checking and test-data generation [1, 2]. Moving the other way round still has its limitations [3]. The reason is the superficial analysis of the specification. As it is not straight forward to identify dependencies within specifications, only simple static class diagrams (that represent state variables of the specification) are generated.

However, since control- and data- dependencies can be identified by specification transformations [4], much more is possible: slices and chunks efficiently allow to excerpt pieces of the specification text with well-defined semantics [5], and cluster generation allows for carving out higher level specification concepts [6]. With this it is also feasible to overcome the limitations mentioned above and to integrate these findings into semi-formal diagrams.

This contribution is structured as follows. Chapter 2 explains the need for bridging the gaps in more detail and presents the state-of-the-art of transforming UML diagrams to formal specifications and vice-versa. It gives special attention to some limits of existing approaches: the scaling problem, and missing dynamics. Chapter 3 discusses ways to identify relevant elements within specifications which will then be the basis for control- and data-dependency reconstruction. Chapter 4 then presents rules for the transformation (based on these dependencies). It explains the approach by making use of a small Z specification. The paper concludes with a short summary and an outlook.

2 Bridging the Gap

It would be the dream of every maintenance personnel, but neither does a SW-system, in general, adopt itself to changing situations or domains (retaining or even improving its quality), nor is it just straight-forward to produce new software products of high quality. As Glass [7, p.122] points out, comprehending the requirements and valid (and consistent) documentation is essential, but rarely all documents are available or are of suitable quality. This is one of the main problems during software comprehension[1]. A situation that should be improved whenever possible.

2.1 Comprehension Challenges

There are several models describing how to maintain software systems [8, 9], but all of them stress that it is necessary to first comprehend the requirements *and* the relevant parts of the underlying system. Starting from scratch and stepping through the code is very time-consuming. Banker et. al. point out the fact that the time needed to comprehend a system on the basis of software code alone is about 3.5 times longer than comprehending the system by additionally studying its documentation [10]. Thus making use of design documents and specifications helps in saving time. But where are the problems?

[1] According to Glass it is second only. The main cause for software comprehension problem is staff turnover.

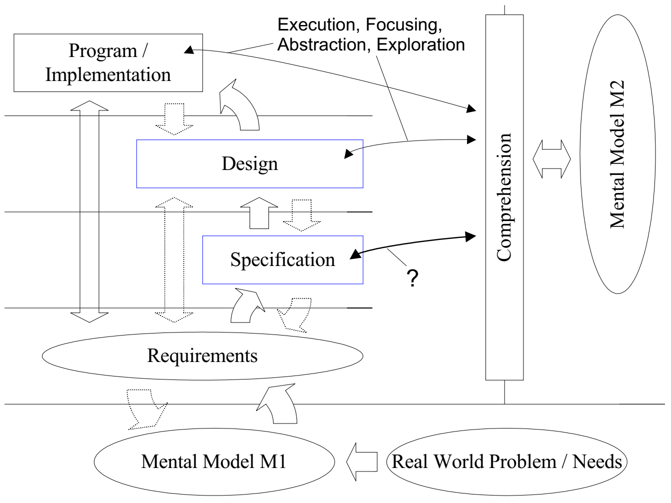

Fig. 1. Software comprehension is an essential process. Besides design documents formal specifications are valid sources for development and maintenance phases. However, to understand formal specifications needs special skills.

Fig. 1 presents a simple (Waterfall-like) model of a development process. It stresses the fact that a mental model $M2$ is reconstructed by an activity called software comprehension – and ideally $M2$ is quite similar to the original model $M1$, which represents the problem and needs. Jackson explains in [11] why formal specifications are the ideal connecting link between the problems of the world (model $M1$ in Fig. 1) and the problems to be solved by the system (the program or implementation). Formal specifications are closer to the requirements – but comprehension needs special skills, and so stakeholders seem to shy away from them. The problems are manifold and are the result of the following gaps between the two worlds:

P1 Formal specifications are said to be of high perceived complexity.
P2 Not all stakeholders that are forced to comprehend the systems (and then to decide) are able to read and understand the underlying documents.
P3 Relating formal specifications with well defined semantics to less formal notations is a loss in precision.
P4 Creating formal specifications from less formal documents is impeded due to information deficiencies. It needs effort to fill the gaps.

What is all to easily overseen is that comprehending a system means the reconstruction of the missing documentation anyway. Concerning problem P1, the overall (and inherent) complexity cannot be reduced. But there are approaches to deal with the density of specifications [12]. And the remaining challenges P2 to P4 (understandability and formality) can be dealt with by consciously mapping the relevant representations to each other. The gaps can be bridged.

2.2 Impediments

The statements so far lead to one observation: the better and more extensive the documentation the easier the comprehension process. A less formal or less mathematical document can also serve its purpose. And a further improvement to the situation is to combine the formal and semi-formal documentation, to reconstruct parts of the documentation, and to be able to switch between different types of notation as needed.

As explained below complexity (problem P1) does not make difficulties. In fact it is in the nature of formal specifications, and it concerns two aspects:

1. Usually there are too few clues for reconstructing the original structure. Putting too much structure into a formal specification is understood to be a hint towards implementation, something that is avoided at the time of writing the specification.
2. The declarative nature also impedes the reconstruction of the behavior. There is no execution-sequence, which is known from programming languages.

Above all, the latter aspect is crucial. As there is no execution order and as there are, in general, no control statements, control- and data-dependency are not predominant concepts (though, as we will see, they are there). Well-known techniques from the field of program comprehension cannot be applied. A formal specification merely covers static information, and alternative forms of representation are then limited to these static information, too.

2.3 Related Work

For the reasons mentioned in Section 2.2 the mapping between formal specifications and less formal approaches is limited to static information. Besides formal extensions to existing notations (e.g. VDM-link to UML [13], or Petri-nets with Z extensions [14]) two directions of the mapping are possible.

Firstly, there is a mapping between some graphical notations to formal specifications. As UML is wide-spread, most of them take static UML diagrams and generate formal state descriptions of it (e.g. UML to Z [15, 16], or UML to Z++ [17, 18]). The approaches have in common that formal specification skeletons are generated which then have to be completed by the designer. As a second step the resulting predicates are simplified, leading, finally, to a full and compact formal specification. This means that semantics has to be added by the designer, but when dealt with it carefully, the specification can be taken to prove properties of the system. Results can then be mapped back to the design documents and deficiencies eliminated.

The other way round is the mapping of formal specification to some graphical notation. An early approach is the Z visualization of Kim [19], who makes use of constraint diagrams (a notation formally defined by [20]). The notation is able to express predicate logic, but there is no integration into existing frameworks. And it is not UML, which does not really ease the understanding among some stakeholders. However, not all the time the full content has to be imparted, and UML, keeping on spreading, is a good target for the transformation.

The approach of Fekih et.al maps B specifications to UML [21]. It takes the state space of the specification and creates an UML class for every abstract set that is element in the domain of relations. The transformation rules are simple but lead to incomplete class diagrams as operations are not regarded. In addition to that the generated classes are not associated. Idani and Ledru improve the approach by mapping occurring relations to UML associations [3]. Furthermore they take operations into account and provide an algorithm for mapping an operation as a method to the most suitable class. Altogether this leads to a more complete static UML diagram.

Contributions mapping formal specifications to UML diagrams follow a pragmatic approach: sets do correspond to classes and relations do correspond to associations. However, as also noted in [3], the resulting diagrams provide less information than the formal specification, and dynamics is not touched at all.

3 Theoretical Background

The reconstruction of dependencies within declarative specification languages is not straight-forward. When looking for control constructs (which will then be the basis for the reconstruction of dynamic behavior), one has to be careful about the basic elements the control is defined about.

3.1 Specification Primes

Specifications are constructed from basic (atomic) units, called specification literals. They can be identified by looking at the grammar of the specification language. As an example, the Z predicate "$assigned \subseteq Permitted$", the specification literals are "*Assigned*", "$\subseteq$", and "*Permitted*". Specification literals are not very expressive when standing alone. It is the combination of literals that makes them rich in content. By aggregating specification literals, *prime objects* of a specification are built.

Definition 1 *A specification prime object represents the basic entity of a specification. It is built out of specification literals and forms logical, syntactic, or semantic units.*

In specification languages these prime objects can be expressions or predicates, but they can also be generic type or schema type definitions. When looking at the decoration of identifiers ($\boxed{\text{'}}$ or $\boxed{!}$ for after-states), post condition primes can easily be identified.

Definition 2 *A Z-specification prime p is considered a post-condition prime, if prime p is an after state invariant. Otherwise it is considered a pre-condition prime.*

The two following predicates of the *AssignResource* operation schema in the *AccessControl* Z Specification (see App. A) can be assigned to one pre- and one post-condition prime,

$$\begin{array}{ll} user? \notin \operatorname{dom} Assigned & \%Precondition\ Prime \\ Assigned' = Assigned \cup \{user? \mapsto resource?\} & \%Postcondition\ Prime \end{array}$$

as the second prime contains an after-state identifier ($Assigned'$). The identification of control-dependencies is then based on these definitions.

Schema	Approximation via Conditions	Related Primes
S	$post\ S \rightrightarrows_c\ pre\ S$	$po_S \rightrightarrows_c pr_S$
$\neg\ S$	$post_s(\neg\ S) \rightrightarrows_c\ pre_w(\neg\ S)$	$po_S \rightrightarrows_c pr_S$
$S \vee T$	$post(S \vee T) \rightrightarrows_c\ pre(S \vee T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \Rightarrow T$	$post_s(S \Rightarrow T) \rightrightarrows_c\ pre_w(S \Rightarrow T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \wedge T$	$post(S \wedge T) \rightrightarrows_c\ pre_w(S \wedge T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \Leftrightarrow T$	$post_s(S \Leftrightarrow T) \rightrightarrows_c\ pre_w(S \Leftrightarrow T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \upharpoonright T$	$post(S \upharpoonright T) \rightrightarrows_c\ pre_w(S \upharpoonright T)$	$(po_S \cup po_T) \rightrightarrows_c (pr_S \cup pr_T)$
$S \fatsemi T$	$post_s(S \fatsemi T) \rightrightarrows_c\ pre(S \fatsemi T)$	$(po_S \cup po_T) \rightrightarrows_c pr_S$
$S \gg T$	$post_s(S \gg T) \rightrightarrows_c\ pre(S \gg T)$	$(po_S \cup po_T) \rightrightarrows_c pr_S$

Table 1. The table summarizes the control dependencies occurring between pre- and postcondition primes in Z schema operations.

3.2 Dependencies

The approach is based on the following simple idea: Preconditions determine whether predicates in the postcondition part are evaluated or not. Thus in specifications postconditions are control dependent on pre-conditions, and the problem of the identification of control dependencies is reduced to the problem of the identification of pre- and post-conditions.

Not all specification languages do make pre- and post-conditions explicit. Additionally, when schema operations are used (see Tab. 1) pre- and post-conditions have to be calculated by performing a semantic analysis, a time- and resource-consuming task which cannot be fully automated. In [12] a syntactic approximation to the semantic analysis is suggested. To summarize: it can be show that precondition primes are at least part of the precondition, and postcondition primes are part of the specifications' postcondition.

Definition 3 *Let S be a schema of a Z specification. Furthermore, let pr_S be the set of pre-condition primes of S and po_S the set of post-condition primes of S. There is control dependency ($po_S \rightrightarrows_c pr_S$) within schema S, if pr_S and po_S are not empty.*

Table 1 summarizes the dependencies among the predicates in the schemata. When there is flow of control data-dependencies are easily detected:

Definition 4 *A specification prime p is data dependent on a specification prime q ($p \rightrightarrows_d q$) if (i) there exists at least one identifier v (literal denoting a data element) that occurs in both p and q, and (ii) v is defined in q and used in p, and (iii) either p and q are in the same scope, or p is control dependent on q.*

4 Semi-Formal Transformation

With the reconstruction of dependencies within declarative specifications it gets possible to exploit both, static *and* dynamic diagrams of UML. In the following a simple Z specification of an Access Control system (see App. A) is taken to illustrate the transformation process.

Type	Symbol	A	B	Type	Symbol	A	B
Relation	$A \leftrightarrow B$	*	*	Part. Surj.	$A \psurj B$	1..*	0..1
Partial	$A \pfun B$	*	0..1	Total Surj.	$A \twoheadrightarrow B$	1..*	1
Total	$A \rightarrow B$	*	1	Total Bij.	$A \bij B$	1	1
Part. Inj.	$A \pinj B$	0..1	0..1	Total Inj.	$A \rightarrowtail B$	0..1	1

Fig. 2. According to [3], relations between sets A and B are mapped to associations with given multiplicities.

4.1 Static Diagrams

The mapping to static class diagrams is based on the idea of looking for global type definitions. It follows mainly the approach of Idani et. al. [3], but omits assigning the operations to derived (and hard to find pertinent) classes. Instead it introduces a system class and assigns the operations to it.

Rule 1 *Every state schema corresponds to a root class with the stereotype <<system>> and the name of the schema.*

Rule 2 *Given sets in the Z specification correspond to classes in the UML specification. They are connected to the system class by using a "use" association.*

Typically this exactly is the view of a specification's designer: having a state and relevant operations. Abstract types are mapped to classes, and these classes are associated to the system class.

A specification also consists of several identifiers holding the state. When they describe relations between given sets, then they are resolved as associations. Subset relations ($\subset$ or $\subseteq$) are made explicitly. Finally, operation names are added, and for the description of the diagram unique names for the associations are introduced:

Rule 3 *Every variable representing relationships between entities in the state schema is translated to associations. It holds that (i) multiplicity is resolved by the mapping rules presented in Fig. 2, and (ii) subsets between relations are resolved by a subset constraint.*

Rule 4 *Every variable representing a subset of entities in the state schema is translated to a specialization class and connected to its root class.*

Rule 5 *Associations do get role-names. They are built by combining the first characters of the source class and association name.*

Rule 6 *Every operation schema is added as a method to those system root class, which has been included in the operations' declaration part. The initialization schema is mapped as a constructor to this class.*

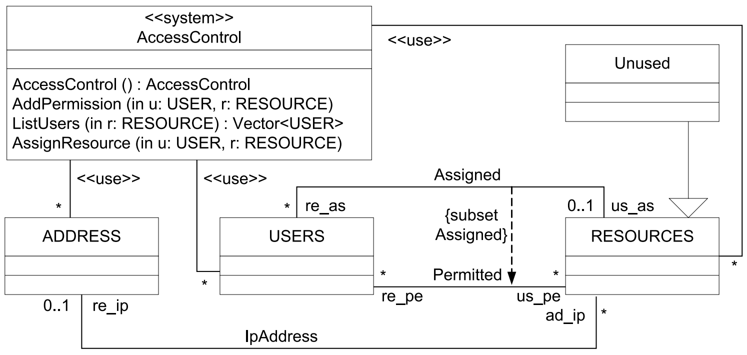

Fig. 3. Applying Rules 1 to 6 leads to a static UML Diagram for the Access Control system specification (see Appendix A).

Fig. 3 presents the result of the transformation of the access control specification. The system class contains the operations. As there are three given set definitions (Users, Resources, Addresses), three classes are introduced and connected to the system class. *Unused* is modelled as a subset of *Resources*, and *Assigned* and *Permitted* are enriched by a subset constraint.

In fact, the above algorithm leads to class/attribute candidates. Still some problems with the transformation remain. It works well when there is only a small set of given sets. With larger specifications the approach of taking given sets as classes leads to a huge static UML diagram. Carving out clusters and concepts [6] and grouping them into extra diagrams might be a way out.

4.2 Dynamics

UML also allows for dynamic diagrams, and as there is also some sort of dynamics within specifications it should be represented in a convenient way. The approach makes use of UML's activity diagrams and focusses on the above defined notion of control and data dependencies.

Rule 7 *Every schema operations is represented by an activity diagram. Logical operations are resolved as boolean expressions. Sequential operations are transformed to sequential control arcs.*

Rule 8 *Control dependencies between two operations are mapped to activity diagrams with control flow vertices. Data dependencies are mapped to activity diagrams by using object flow nodes with the label of the relevant identifier(s).*

Our specification has one schema operation called *ActiveAccessControl*. It contains a sequential schema operation and the logical combination of the remaining operation schemata. Fig. 4 (left side) presents the result of applying rule 7. It puts the initialization in sequence to the operations, which are logically combined by an OR-operation.

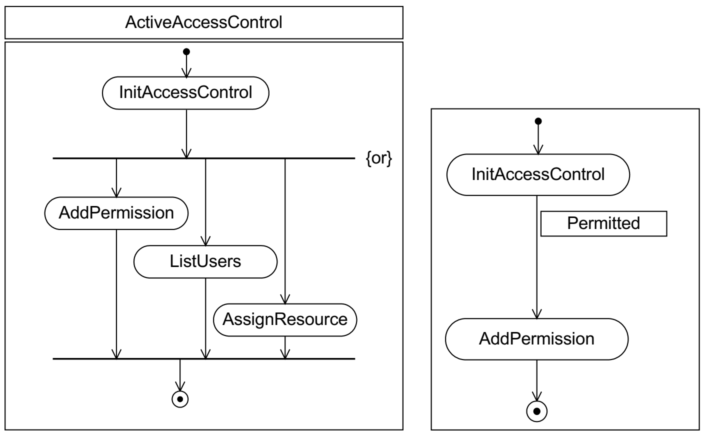

Fig. 4. Applying rules 7 and 8 leads to two UML Activity Diagrams for the Access Control system specification (see Appendix A).

According to the definition of control dependency and the rules in Tab. 1 there is, e.g, control dependency between the two schemata *AccessControl* and *AddPermission* (as *AddPermission* includes the state schema and has post-condition primes which get dependent on the *AddPermission*'s pre-condition primes). Therefor another diagram is added, showing the control dependency by an dependency arc. Due to the fact that there is also data-dependency (the value of *Permitted* is relevant), *Permitted* is added as an object flow node.

Finally, the more formal part of the specification must not be forgotten. By using OCL constraints, the relevant pre- and post-condition predicates can be included.

Rule 9 *Pre- and Post-conditions of operation schemata are mapped to pre and post OCL comments to the system class; the predicates of the state schema are mapped to the class as OCL invariants. For Z operations expressive function names are to be chosen, their semantics is to be explained in a legend box.*

The OCL comments can, of course, be left out of account (depending on the stakeholder). However, by the above rule almost all relevant parts of the specification have been mapped. What remains are some open issues.

Fig. 5 demonstrates the result of the transformation to a set of OCL comments. The constructor (initialization schema) gets a postcondition constraint, the three operation schemata get a pre- and post-condition each. The post-condition of *ListUsers*, e.g., tells us that the application does not change the set of resources ($r == r \bullet pre$, the after-state is the same as the before state). The system class is associated with the class invariant. Typically as is, predicate logic, Z functions, and operations are difficult to express in OCL. Here it is suggested to choose mnemonic names for functions and to explain them, as needed, in an extra legend box.

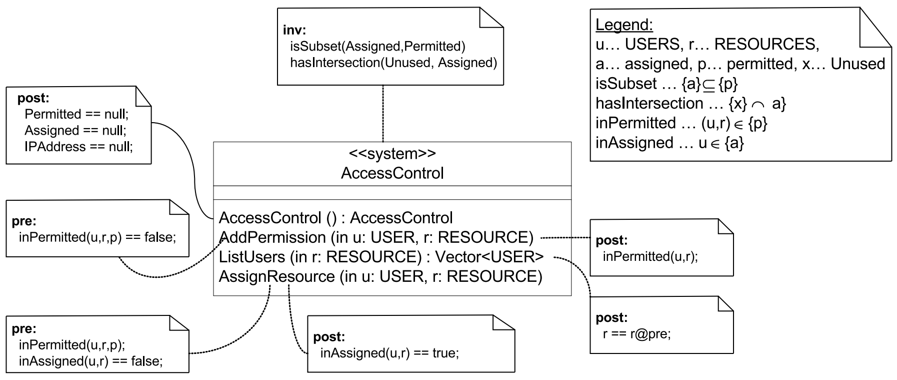
inv:
isSubset(Assigned,Permitted)
hasIntersection(Unused, Assigned)
Legend:
u… USERS, r… RESOURCES,
a… assigned, p… permitted, x… Unused
isSubset … {a}⊆{p}
hasIntersection … {x} ∩ a}
inPermitted … (u,r) ∈ {p}
inAssigned … u ∈ {a}
post:
Permitted == null;
Assigned == null;
IPAddress == null;
<<system>>
AccessControl
AccessControl () : AccessControl
AddPermission (in u: USER, r: RESOURCE)
ListUsers (in r: RESOURCE) : Vector<USER>
AssignResource (in u: USER, r: RESOURCE)
pre:
inPermitted(u,r,p) == false;
post:
inPermitted(u,r);
post:
r == r@pre;
pre:
inPermitted(u,r,p);
inAssigned(u,r) == false;
post:
inAssigned(u,r) == true;

Fig. 5. Applying rule 9 leads to an annotated static UML Diagram that makes use of OCL-like annotations. A set of "self-explaining" function names are used in order to represent Z functions that cannot be mapped to OCL expressions.

5 Conclusion

Comprehension is an inevitable task during software maintenance and development phases, and specifications, when kept up-to-date, are valid sources. However, due to their complexity, it is not surprising that formal specifications languages are said to be write-only languages.

This paper discusses ways in transforming formal specifications to UML in order to open the documents to a wider range of stakeholders. Existing approaches only cover statics, but state-based specifications also deal with state changes – and thus dynamics. In contrary to existing approaches (which focus on class diagrams) this paper suggests to make also use of activity diagrams. It explains how to identify control dependencies, which are then the basis for the latter representation of dynamic behavior within declarative specifications.

There are still some limitations that should not be concealed. As with other approaches the issue of scalability is not solved, and in addition to that it is still hard to decide whether a class candidate is a pertinent class or not. It might be helpful to map these candidates to the concepts identified by cluster generation [6], and then to decide whether to accept them or not (as they match to specific clusters). To test this applicability a framework for dealing with larger Z specifications, combining slicing, chunking, and clustering facilities is under further development.

The approach does, by far, not lead to a perfect UML representation of the specification. But it provides a good picture of *what* is *in* the specification. In fact it can at least be used to speed up the construction of concepts behind. And as dynamics is at least as important as statics, this approach should be a step further into the direction of a more useable framework and increase the use of formal specification.

References

1. Laleau, R., Mammar, A.: An overview of a method and its support tool for generating B specifications from uml notations. In: Fiftheenth IEEE Conference on Automated Software Engineering. (2000)
2. Truong, N.T., Souquieres, J.: An approach for the verification of UML models using B. In: Proceedings of the 11th IEEE Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04). (2004)
3. Idani, A., Ledru, Y.: Object oriented concepts identification from formal B specifications. In: Formal Methods in Industrial Critical Applications, FMICS'04. (2004)
4. Mittermeir, R.T., Bollin, A.: Demand-driven specification partitioning. In: Proceedings of the 5th Joint Modular Languages Conference, JMLC'03. (2003)
5. Bollin, A.: Maintaining formal specifications. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary. (2005) 442–453
6. Bollin, A.: Concept location in formal specifications. In: Submitted to the Intl. Conference on SW-Maintenance 2006. (2006)
7. Glass, R.L.: Facts and Fallacies of Software Engineering. Addison-Wesley (2003)
8. Basili, R.V.: Viewing maintenance as reuse-oriented software development. IEEE Software **7**(1) (1990) 19–25
9. Pirker, H.: Specification based Software Maintenance (a Motivation for Service Channels). PhD thesis, University of Klagenfurt (2001)
10. Banker, R.D., Davis, G.B., Slaughter, S.A.: Software development practices, software complexity, and software maintenance performance: A field study. In: Management Science. Volume 44., Inst. for Operations Research and the Management Sciences (1998) 433–450
11. Jackson, M.: The world and the machine. In: Proc. 17th International Conference on Software Engineering, IEEE-CS Press (1995) 283–292
12. Bollin, A.: Specification Comprehension – Reducing the Complexity of Specifications. PhD thesis, University of Klagenfurt (2004)
13. Dick, J., Loubersac, J.: A visual approach to VDM: Entity-structure diagrams. Technical Report DE/DRPA/91001, Bull, 68, Route de Versailles, 78430 Louveciennes (France) (1991)
14. He, X.: PZ Nets - a formal method integrating Petri Nets with Z. Information and Software Technology **43**(1) (2001) 1–18
15. Dupuy, S., Ledru, Y., Chabre-Peccoud, M.: An overview of RoZ: A tool for integrating UML and Z specifications. In: Proceedings of CAiSE'00. (2000) 417–430
16. Idani, A., Ledru, Y., Bert, D.: Derivation of UML class diagrams as static views of formal B developments. In: 7th International Conference on Formal Engineering Methods, ICFEM 2005. (2005) 37–51
17. Kim, S.K., Carrington, D.: A formal mapping between UML models and Object-Z specifications. Lecture Notes in Computer Science **1878** (2000) 2–21
18. Roe, D., Broda, K., Russo, A.: Mapping UML models incorporating OCL constraints into Object-Z. Technical Report ISBN/ISSN: 1469-4174, Imperial College of Science, Technology and Medicine, Department of Computing (2003)
19. Kim, S.K., Carrington, D.: Visualization of formal specifications. In: In Proceedings Sixth Asia Pacific Software Engineering Conference (ASPEC'99), IEEE Computer. Society Press, Los Alamitos, CA, USA (1999) 102–109
20. Kent, S.: Constraint diagrams: Visualising invariants in object-oriented models. In: In Proceedings of OOPSLA'97, ACM Press (1997)
21. Fekih, H., Jemni, L., Merz, S.: Transformation des spècifications B en des diagrammes UML. In: Approches Formelles dans l'Assistance au Dveloppement de Logiciels, AFADL'04. (2004) 131–148

Appendix A - Access Control Specification

$$[\mathit{USERS}, \mathit{RESOURCES}, \mathit{ADDRESSES}]$$

$$\begin{array}{|l}
\hline
\mathit{AccessControl} \\
\hline
\mathit{Permitted} : \mathit{USERS} \leftrightarrow \mathit{RESOURCES} \\
\mathit{Assigned} : \mathit{USERS} \mathrel{+\!\!\!\!\rightarrow} \mathit{RESOURCES} \\
\mathit{IpAddress} : \mathit{ADDRESSES} \mathrel{+\!\!\!\!\rightarrow} \mathit{RESOURCES} \\
\mathit{Unused} : \mathbb{P}\,\mathit{RESOURCES} \\
\hline
\mathit{Assigned} \subseteq \mathit{Permitted} \\
\mathit{Unused} \cap (\operatorname{ran} \mathit{Assigned}) = \varnothing \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{InitAccessControl} \\
\hline
\mathit{AccessControl} \\
\hline
\mathit{Permitted} = \varnothing \\
\mathit{Assigned} = \varnothing \\
\mathit{IpAddress} = \varnothing \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{AddPermission} \\
\hline
\Delta \mathit{AccessControl} \\
\mathit{user}? : \mathit{USERS} \\
\mathit{resource}? : \mathit{RESOURCES} \\
\hline
(\mathit{user}? \mapsto \mathit{resource}?) \notin \mathit{Permitted} \\
\mathit{Permitted}' = \mathit{Permitted} \cup \{\mathit{user}? \mapsto \mathit{resource}?\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{ListUsers} \\
\hline
\Xi \mathit{AccessControl} \\
\mathit{resource}? : \mathit{RESOURCE} \\
\mathit{st}! : \mathbb{P}\,\mathit{USERS} \\
\hline
\mathit{st}! = \operatorname{dom}(\mathit{Permitted} \rhd \{\mathit{resource}?\}) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{AssignResource} \\
\hline
\Delta \mathit{AccessControl} \\
\mathit{user}? : \mathit{USERS} \\
\mathit{resource}? : \mathit{RESOURCES} \\
\hline
(\mathit{user}? \mapsto \mathit{resource}?) \in \mathit{Permitted} \\
\mathit{user}? \notin \operatorname{dom} \mathit{Assigned} \\
\mathit{Assigned}' = \mathit{Assigned} \cup \{\mathit{user}? \mapsto \mathit{resource}?\} \\
\hline
\end{array}$$

$$\begin{aligned}
\mathit{ActiveAccessControl} == {} & \mathit{InitAccessControl} \mathbin{⨟} \\
& (\mathit{AddPermission} \lor \mathit{ListUsers} \lor \mathit{AssignResource})
\end{aligned}$$