

Software Certification Management How Can Formal Methods Help?

Dieter Hutter

German Research Centre for Artificial Intelligence (DFKI GmbH),
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany,
e-mail: hutter@dfki.de

Abstract

The formal development of industrial-size software is an error-prone and therefore evolutionary process. We report on our efforts to implement an assistance tool that helps us to anticipate the effects of changes in formal specification, to retrieve existing specifications and adapt them to new situations, to determine the “minimal” sets of proof obligations that will newly arise or which proofs have to be re-tackled again, and to adjust the old proofs to the new conditions. As a result of this work we outline an underlying theoretical framework for a general repository to maintain mathematical or logic-based documents while keeping track of the various semantical dependencies between different parts of various types of documents (documentations, specifications, proofs, etc).

1 Introduction

In the last decade Formal Methods have been successfully applied to specify and verify security or safety critical systems. In the area of security, the formal specification (and partly also verification) of smartcards became a necessity to comply with the security requirements of their users. Car manufacturers start to use formal methods to get the more and more complex devices and sophisticated interaction between them under control. Formal software development paradigms are closely related to the waterfall model.

Starting with a formal (textual) specification, it is translated into a logic based formalism, proof obligations are calculated to guarantee the security or safety properties, and finally these obligations have to be proven usually with the help of model checking or theorem proving. However in all applications so far, development steps turned out to be flawed and errors had to be corrected. The search for formally correct software and the corresponding proofs is more like a formal reflection on partial developments rather than just a way to assure and prove more or less evident facts. Figure 1 illustrates this typical process.

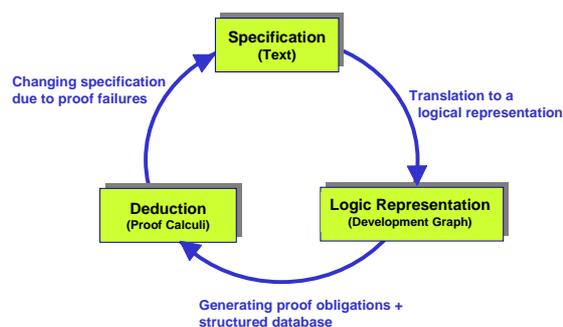


Figure 1: Formal Development Cycle

2 Development Graphs

The painful experience of this evolutionary character of applying formal methods resulted in the development of the MAYA system [2, 3] to maintain the formal software development process. We envisioned an assistance tool that helps us to anticipate the effects of changes in the specification, to retrieve old specifications and adapt them to new situations, to determine the “minimal” sets of proof obligations that will newly arise or which proofs have to be re-tackled again, and to adjust the old proofs to the new conditions.

MAYA was a first step towards such a system. It supports an evolutionary formal development since it allows users to specify and verify developments in a structured manner, incorporates a uniform mechanism for verification *in-the-large* to exploit the structure of the specification, and maintains the verification work already done when changing the specification. MAYA relies on development graphs as a uniform representation of structured specifications, which enables the use of various (structured) specification languages like CASL [4], OMDOC and VSE-SL [6] to formalize software development. To this end MAYA provides a generic interface to plug in additional parsers for the support of other specification languages. Moreover, MAYA allows the integration of different theorem provers to deal with proof obligations arising from the specification, i.e. to perform verification *in-the-small*.

Textual specifications are translated into a structured logical representation called a *development graph* [1, 5], which is based on the notions of consequence relations and morphisms and makes arising proof obligations explicit. The user can tackle these proof obligations with the help of theorem provers connected to MAYA like Isabelle [12] or INKA [8].

A failure to prove one of these obligations usually gives rise to modifications of the underlying specification. MAYA supports this evolutionary development process as it calculates minimal changes to the logical representation readjusting it to a modified specification while preserving as much verification work as possible. If necessary it also adjusts the database of the interconnected theorem prover. Furthermore,

MAYA communicates to the attached provers explicit information how the axiomatization has changed and also retrieves former proofs of the same problem (that are now invalidated by the changes) to allow the theorem provers to reuse them. In turn, information provided by the theorem provers about the computed proof is used to optimize the maintenance of proofs during the evolutionary development process.

3 Transformational Development

While MAYA supports the adaption of formal developments to changed specifications, it primarily focuses on the computation of differences between old and new specification and to maintain and propagate these changes along the development cycle illustrated in Figure 1. However, due to the complexity of this process there is no guarantee that “simple” changes in the specification will be adequately supported by the system when it comes to the adaptation of proofs. Starting with [14] we worked also on defining building blocks to transform developments as a whole. The idea is to incorporate the knowledge about the way a specification is changed into rules how to adapt the existing proofs simultaneously. This results in a set of basic transformations operating on developments and changing specifications *together* with their proof obligations and proofs in parallel and returning an adapted new development. Typical examples are the change of abstract datatypes by adding or removing constructors, the change of parameters of function and predicates, and the change of axioms by adding or removing conditions [13]. Using these basic transformations allows a developer to predict the effects of his changes to the entire development since each of these basic transformations will change specification, proofs and proof obligations in a predetermined and controlled fashion.

4 A More General Approach

Developing the MAYA system, which is specialized to the maintenance of formal developments based on

algebraic specifications, we realized that the underlying methodology is rather general and independent of the used logical representation of specifications and proofs but relies heavily on the structure of dependencies between objects and properties and how these dependencies can be decomposed along a given structure.

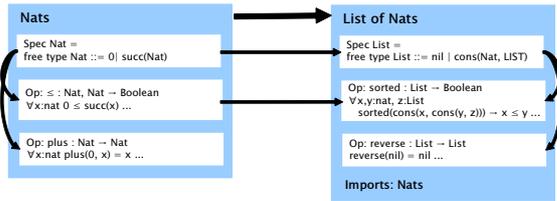


Figure 2: Semantic Dependencies in Specifications

In formal developments the semantics of (structured) objects depends on the semantics of their sub-objects used for their definition (axiomatic dependencies). While there is no way to scrutinize changes of axiomatic dependencies in case of intentional (or purposeful) changes of the development, there is a need to inspect axiomatic dependencies in case of mechanical changes as they occur, for instance, during the (automatic) merge of two branches of a development.

Properties between (structured) objects can be postulated and proven within a development. Similar to the objects under consideration, the proofs of properties about such objects are also structured. Hence, such a proof depends on (or decomposes into) properties of sub-objects which gives rise to *deduced dependencies* between different properties. Changing the development may render proofs invalid since either some basic property does no longer hold or the way the problem was decomposed is no longer appropriate.

The ability to decompose properties along the structure of the concerned objects allows us to localize the effects of changes. A property between structured objects (theories) is decomposed (according to some decomposition rules) to properties between their sub-objects (local axioms). Typically, these properties between structured objects have to

be independent of the environment in which these objects might occur. As long as the concerned structured objects are unchanged any change of the overall development will not inflict the already proven or postulated properties between these objects.

5 Repository Supporting Distributed Development

As a consequence we now work on a repository [7] to maintain all sorts of dependencies between various parts of a formal development or even informal documents [9]. The main goal of such a logic-based repository is to ease the development of mathematical or logic based knowledge consisting of entities such as axioms, definitions, theorems, proofs and informal documentations (sometimes including semantic annotations). As the development of a software project or of mathematical knowledge is distributed, also the repository has to support distributed developments. We propose a CVS-like infrastructure to determine the differences between two versions, to calculate the necessary changes to update a local repository to the current state, and to integrate two rival developments into a merged variant. However while text-lines might be appropriate to structure pure text documents, this approach fails completely in logic-based documents. A single text line may contain independent terms or a single term could be spread over many text lines. Undiscovered “semantic” conflicts may occur if two users change different text lines that are both part of the description of a single term. Changing the arity of a signature symbol in a document typically requires to change its arity in all the occurrences of the symbol. Therefore, we use the more semantically adequate representation of acyclic directed graphs as the general structure underlying the documents under consideration and redefine the CVS notions of diff, patch and merge in this context.

In a second phase we add semantical dependencies between different parts of a document to detect semantical conflicts, for instance, when merging different versions of a document that result from chang-

ing different but semantically still dependent parts of a document by different users. When merging documents, semantic conflicts occur if semantically related documents are changed independently by two users. Decomposing dependencies along the structure of the objects allows one again to narrow down potential semantic conflicts: conflicts of composed objects only arise if there is a conflict between dependent sub-objects.

6 Conclusion

Inspecting the ideas of MAYA we discovered that most of the work related to the management of change does not require a deep knowledge of the semantics of the underlying specification languages. Instead the management of change solely operates on the structure of the objects under consideration and on how proposed properties can be decomposed to properties of their sub-objects.

The ultimate goal is to support generic structuring mechanisms as they occur in various domains by developing a system supporting these mechanisms while outsourcing application specific parts into modules attachable to the system. This would allow us to instantiate such a system for various purposes, like for instance in formal methods (cf. MAYA [2]), program development, or even maintaining informal documents like, for instance, course materials (cf. MMISS [9]).

References

- [1] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. In *Recent Developments in Algebraic Development Techniques, WADT'99, Bonas, France*, Springer LNCS 1827, 2000.
- [2] S. Autexier, D. Hutter, T. Mossakowski and A. Schairer. The Development Graph Manager MAYA. In *Proceedings 9th International Conference on Algebraic Methodology And Software Technology, AMAST2002*. Springer, LNCS 2422, 2002
- [3] S. Autexier and D. Hutter. Mind the Gap - Maintaining Formal Developments in MAYA, In *Mechanising Mathematical Reasoning, Essays in honor of J.H. Siekmann*, Springer-Verlag, LNCS 2605, 2005
- [4] B. Krieg-Brückner and P. Mosses (eds). CASL Reference Manual, Springer, LNCS 2960, 2004
- [5] D. Hutter. Management of Change in Verification Systems. In *Proceedings 15th IEEE International Conference on Automated Software Engineering, ASE-2000*, IEEE Computer Society, 2000.
- [6] D. Hutter et al. Verification Support Environment (VSE), *Journal of High Integrity Systems*, Vol. 1, pages 523–530, 1996.
- [7] D. Hutter. Towards a Generic Management of Change. In *Workshop on Computer-Supported Mathematical Theory Development, International Joint Conference on Automated Reasoning'04*, Cork, Ireland, 2004
- [8] S. Autexier, D. Hutter, H. Mantel, A. Schairer: System Description: INKA 5.0 - A Logic Voyager. In H. Ganzinger, *CADE-16*, Springer, LNAI 1632, 1999.
- [9] B. Krieg-Brückner, D. Hutter, C. Lüth, E. Melis, A. Pötsch-Heffter, M. Roggenbach, J. Smaus and M. Wirsing. Towards MultiMedia Instruction in Safe and Secure Systems. In: *Recent Trends in Algebraic Development Techniques, (WADT-02)*. Springer, LNCS 2755, 2003
- [10] T. Mossakowski and P. Hoffman and S. Autexier and D. Hutter. Part IV: CASL Logic. In: [4], 2004
- [11] T. Mossakowski, S. Autexier, and D. Hutter. Development Graphs – Proof Management for Structured Specifications. *Journal of Logic and Algebraic Programming*, Special Issue on Algebraic Specification and Development Techniques, Elsevier, 2005 (forthcoming)
- [12] L.C. Paulson. *Isabelle - A Generic Theorem Prover*, Springer LNCS 828, 1994.
- [13] A. Schairer. Transformations of Specifications and Proofs to Support an Evolutionary Formal Software Development. PhD thesis (submitted), Saarland University, 2005
- [14] A. Schairer and D. Hutter Proof Transformations for Evolutionary Formal Software Development. In: *Proceedings of the 9th International Conference on Algebraic Methodology And Software Technology, AMAST-2002*, LNCS 2422, 2002