

The Development Graph Manager MAYA

Serge Autexier¹, Dieter Hutter¹, Till Mossakowski², and Axel Schairer¹

¹ DKFI GmbH, Stuhlsatzenhausweg 3, D 66123 Saarbrücken,
{autexier|hutter|schairer}@dfki.de

² FB 3, University of Bremen, P.O. Box 330 440, D 28334 Bremen, till@tzi.de

1 Overview

Formal methods are used in the software development process to increase the security and safety of software. The software systems as well as their requirement specifications are formalised in a textual manner in some specification language like CASL [3] or VSE-SL [10]. The specification languages provide constructs to structure the textual specifications to ease the reuse of components. Exploiting this structure, e.g. by identifying shared components in the system specification and the requirement specification, can result in a drastic reduction of the proof obligations, and hence of the development time which again reduces the overall project costs.

However, the logical formalisation of software systems is error-prone. Since even the verification of small-sized industrial developments requires several person months, specification errors revealed in late verification phases pose an incalculable risk for the overall project costs. An *evolutionary formal development* approach is absolutely indispensable. 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* of partial developments rather than just a way to assure and prove more or less evident facts.

The MAYA-system 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 [3] and VSE-SL [10] to formalise the 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 the actual 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, 4], 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 [8] or INKA [5].

A failure to prove one of these obligations usually gives rise to modifications of the underlying specification (see Fig. 1). MAYA supports this evolutionary 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 explicit information how the axiomatisation has changed and also makes available proofs of the same problem (invalidated by the changes) to allow for a reuse of proofs inside the theorem provers. In turn, information about a proof provided by the theorem provers is used to optimise the maintenance of the proof during the evolutionary development process.

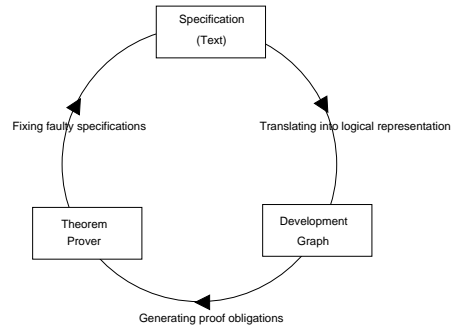


Fig. 1. Formal Life Cycle

2 From Textual to Logical Representation

The specification of a formal development in MAYA is always done in a textual way using specification languages like CASL or VSE-SL. MAYA incorporates parsers to translate such specifications into the MAYA-internal specification language DGRL (“Development Graph Representation Language”). While unstructured specifications are solely represented as a signature together with a set of logical formulas, the structuring operations of the specification languages (such as **then**, **and**, or **with** in CASL) are translated into the structure of a development graph. Each node of this graph corresponds to a theory. The axiomatisation of this theory is split into a local part which is attached to the node as a set of higher-order formulas and into global parts, denoted by ingoing definition links, which import the axiomatisation of other nodes via some consequence morphisms. While a so-called *local* link imports only the local part of the axiomatisation of the source node of a link, *global* links are used to import the entire axiomatisation of a source node (including all the imported axiomatisations of other nodes). In the same way local and global *theorem links* are used to postulate relations between nodes (see [1] for details). A global theorem link represents that all axioms defining the theory of the source node are conjectures in the theory of the target node, while local theorem links represent that only the local axioms of the source node are conjectures in the theory of the target node. Consider, as an example, the CASL-specification of lists of natural numbers and stacks over arbitrary elements (cf. Fig. 2). The **view** postulates that lists over natural numbers are indeed stacks. The specifications are translated into the development graph viewed in the upper left part of Fig. 3. The resulting development graph is structurally more verbose than the original textual specification structure because of the different definitions of scope and visibility for

```

emacs: AusrX11/minXemacs [21.1 (patch 14) "Guyahoga Valley" XEmacs Lucid] AMAST.casl
File Edit Hule Apps Options Buffers Tools Top<<< . >>> Bot Help
Open Undo Save Print Cut Copy Paste Undo Spell Replace Mail Info Complete Debug News
#spec natlist =
{
  generated type nat ::= null | s(p:nat);
  var x,y,z:nat;
  op * : nat * nat -> nat, comm, assoc, unit s(null);
  op +(x:nat; y:nat):nat =
    y when x = null
    else s(+(p(x), y));
  axiom +(x,y) = +(y,x);
  axiom +(x,(y,z)) = +(x,y),z;
}
then
{
  generated type natlist ::= nil
    | cons(fst:nat; rest:natlist);
  var l1,l2:natlist;
  var n1,n2:nat;
  op app : natlist * natlist -> natlist, assoc, unit nil;
  axiom app(cons(n1,l1),l2) = cons(n1, app(l1,l2));
  op adl1ast(n:nat; l:natlist):natlist =
    cons(n,nil) when l = nil
    else cons(fst(l), adl1ast(n.rest(l)));
  op delete_until(n:nat; l:natlist):natlist =
    nil when l = nil
    else rest(l) when n = fst(l)
    else delete_until(n, rest(l));
}
}

#spec stack =
{
  sort elem;
}
then
{
  generated type stack ::= empty_stack
    | push(top:elem; pop:stack);
  op poprec(e:elem; s:stack):stack =
    empty_stack when s = empty_stack
    else pop(s) when e = top(s)
    else poprec(e, pop(s));
}

view viewit : stack to natlist =
  sorts elem |-> nat,
  stack |-> natlist,
  ope poprec:elem * stack -> stack |-> delete_until,
  empty_stack:stack |-> nil,
  top: stack -> elem |-> fst,
  pop: stack -> stack |-> rest,
  push:elem * stack -> stack |-> cons
end

```

Fig. 2. Textual CASL-specification

signature symbols: The visibility rules for the different CASL structuring operations are encoded in the uniform structuring operation (definition links) of the development graph. The left subgraph results from the specification of stacks over arbitrary elements, while the right subgraph corresponds to the lists over natural numbers. The `view` is encoded by a global theorem link from the top node of the left subgraph to the top node of the right subgraph.

3 Verification In-the-large

The development graph is the central data-structure to store and maintain the formal (structured) specification, the arising proof obligations and the status of the corresponding verification effort (proofs) during a formal development.

MAYA distinguishes between proof obligations postulating properties between different theories, like the notion of `view` in CASL or `satisfies` in VSE-SL, and lemmata postulated within a single theory, e.g. with the `%implies` annotation in CASL. As theories correspond to subgraphs within the development graph, a relation between different theories, represented by a global theorem link, corresponds to a relation between two subgraphs. Each change of these subgraphs can affect this relation and would invalidate previous proofs of this relation. Therefore, MAYA decomposes relations between different theories into individual relations between the local axiomatisation of a node and a theory (denoted by a local theorem link). Each of these relations decomposes again into a set of proof obligations postulating that each local axiom of the node is a theorem in the target theory with respect to the morphism attached to the link. In the running example the global theorem link between the top nodes of both subgraphs is

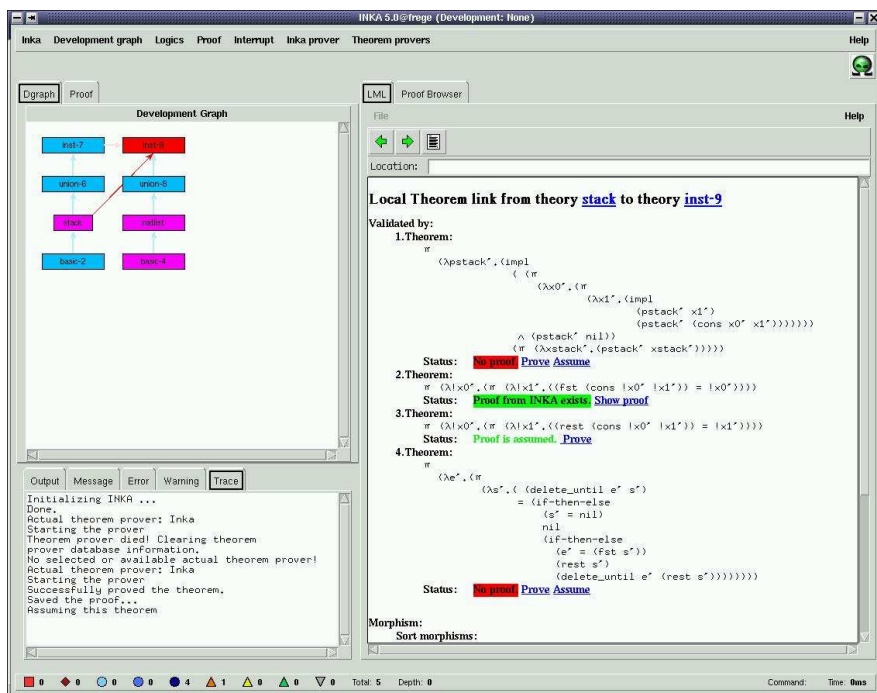


Fig. 3. Bookkeeping the status of proof obligations in MAYA

decomposed into a set of local theorem links from each node of the left subgraph to the top node of the right subgraph (cf. Fig. 3). The proof obligations of such a local theorem link are viewed in the right-hand side of Fig. 3.

While definition links establish relations between theories, theorem links denote lemmata postulated about such relations. Thus, the reachability between two nodes establishes a formal relation between the connected nodes (i.e. the theory of the source node is part of the theory of the target node wrt. the morphisms attached to the connecting links). MAYA uses this property to prove relations between theories by searching for paths between the corresponding nodes (instead of decomposing the corresponding proof obligation in the first place).

4 Verification In-the-small

When verifying a local theorem link or proving speculated lemmata, the conjectures have to be tackled by some interconnected theorem prover. In both cases the proofs are done *within* the theory of a specific node. Thus, conceptually each node may include its own theorem prover which is provided with the local axioms of the node and all axioms imported via incoming definition links. In principle, there is a large scale of integration types. The tightest integration consists of having a theorem prover for each node wrt. which theory conjectures must be proven, and the theorem prover returns a proof object generated during

the proof of a conjecture. Those are stored together with the conjecture and can be used by MAYA to establish the validity of the conjecture if the specification is changed. The loosest integration consists in having a single generic theorem prover, which is requested to prove a conjecture within some theory and is provided with the axiomatisation of this theory. The theorem prover only returns whether it could prove a conjecture or not, without any information about axioms used during the proof. For a detailed discussion of the advantages and drawbacks of the different integration scenarios see [2].

Currently, MAYA supports two integration types: One where information about used axioms is provided by the theorem prover, and one where no such information is provided. In the first case, MAYA stores the proof information and the axioms used during the proof (cf. status of theorem 2 in Fig. 3). In the second case, MAYA assumes there is a proof for the proof obligation (cf. status of theorem 3 in Fig. 3), as there is no information about the proof. In both scenarios, MAYA makes use of generic theorem provers which are provided with the axiomatisation of the current theory. Currently MAYA provides all axioms and lemmata located at theories that are imported from the actual theory by definition links to the prover. Switching between different proof obligations may cause a change of the current underlying theory and thus a change of the underlying axiomatisation. MAYA provides a generic interface to plug in theorem provers (based on an XML-RPC protocol) that allows for an incremental update of the database of the prover.

5 Evolution of Developments

Changes of specifications are done inside the textual representation. Parsing a modified specification results in a modified DGRL-specification. In order to support a management of change, MAYA computes the differences of both DGRL-specifications and compiles them into a sequence of basic operations in order to transform the development graph corresponding to the original DGRL-specification to a new one corresponding to the modified DGRL-specification. Examples of such basic operations are the insertion or deletion of a node or a link, the change of the annotated morphism of a link, or the change of the local axiomatisation of a node. As there is no optimal solution to the problem of computing differences between two specifications, MAYA uses heuristics based on names and types of individual objects to guide the process of mapping corresponding parts of old and new specification. Since the differences of two specifications are computed on the basis of the internal DGRL-representation, new specification languages can easily be incorporated into MAYA by providing a parser for this language and a translator into DGRL.

The development graph is always synthesised or manipulated with the help of the previously mentioned basic operations (insertion/deletion/change of nodes/links/axiomatisation) and MAYA incorporates sophisticated techniques to analyse how these operations will affect proof obligations or proofs stored within the development graph. They incorporate a notion of monotonicity of theories and

morphisms, and take into account the sequence in which objects are inserted into the development graph. Furthermore, the information about the decomposition and subsumption of global theorem links obtained during the verification *in-the-large* is explicitly maintained and exploited to adjust them once the development graph is altered. Finally, the knowledge about proofs, e.g. the used axioms, provided by the interconnected theorem provers during the verification *in-the-small* is used to preserve or invalidate the proofs.

6 Conclusion

The MAYA-system is mostly implemented in Common Lisp while parts of the GUI, shared with the OMEGA-system [9], are written in Mozart. The CASL-parser is provided by the CoFI-group in Bremen. The MAYA-system is available from the MAYA-web-page at www.dfki.de/~inka/maya.html.

Future extensions of the system will include a treatment of hiding [7], a uniform treatment of different logics based on the notion of heterogeneous development graphs [6], and the maintenance of theory-specific control information for theorem provers. The latter comprises a management for building up the database of theorem provers by demand rather than providing all available axioms and lemmata at once as well as the management of meta-level information, like tactics or proof plans, inside MAYA.

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, T. Mossakowski. Integrating HOLCASL into the Development Graph Manager MAYA. In A. Armando (Ed.) *Frontiers of Combining Systems (FroCoS'02)*, Santa Margherita Ligure, Italy, Springer LNAI, April, 2002.
3. CoFI Language Design Task Group. *The Common Algebraic Specification Language (CASL) – Summary, Version 1.0 and additional Note S-9 on Semantics*, available from <http://www.cofi.info>, 1998.
4. D. Hutter. Management of change in verification systems. In *Proceedings 15th IEEE International Conference on Automated Software Engineering, ASE-2000*, pages 23–34. IEEE Computer Society, 2000.
5. 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.
6. T. Mossakowski. Heterogeneous development graphs and heterogeneous borrowing. In M. Nielsen (Ed.) *Foundations of Software Science and Computation Structures (FOSSACS02)*, Grenoble, France, Springer LNCS, April, 2002.
7. T. Mossakowski, S. Autexier, and D. Hutter. Extending development graphs with hiding. In A. Konermann, editor, *Proceedings of Fundamental Approaches to Software Engineering (FASE2001)*. Springer, LNCS 2029, 2001.
8. L.C. Paulson. *Isabelle - A Generic Theorem Prover*, Springer LNCS 828, 1994.
9. J. Siekmann et al. LOUI : Lovely OMEGA user interface. *Formal Aspects of Computing*, 3(11):326-342, 1999.
10. D. Hutter et. al. Verification Support Environment (VSE), *Journal of High Integrity Systems*, Vol. 1, pages 523–530, 1996.