Tool Integration in the UniForM-Workbench*

C. Lüth[1], E. W. Karlsen[1], Kolyang[1], S. Westmeier[1], and B. Wolff[2]

[1] Bremen Institute for Safe Systems, FB 3, Universität Bremen
Postfach 330440, 28334 Bremen
{cxl,ewk,kol,litfox}@informatik.uni-bremen.de
[2] Universität Freiburg, Institut für Informatik
wolff@informatik.uni-freiburg.de

Abstract. The UniForM-Workbench is an open tool integration environment providing type safe communication, a toolkit for graphical user interfaces, version management and configuration management. It is specifically geared towards formal methods and based on the encoding of formal methods into the theorem prover Isabelle. The UniForM-Workbench thus tackles both the technical and the semantical problems arising when integrating formal method tools.

1 Introduction

The need for tool support is widely acknowledged, in particular in the context of formal methods. Yet, there exist many different tools serving different formal methods, and no single formal method (let alone tool) covers all aspects of software development. In order to combine the advantages of various tools and methods, they are combined into one integrated *Software Development Environment* (SDE).

Tool integration for formal methods has to address two issues: the technical side of the integration (where features such as type-safe communication, persistent and distributed storage, version management and configuration management are required), and the semantic side of the integration (where the semantic integrity of the documents produced by the various formal method tools have to be maintained).

The UniForM-Workbench addresses both of these issues. On the technical side, the integration is based on a *loosely coupled* architecture where prefabricated tools are integrated on top of a *tool integration framework*. On the semantic side, we propose an approach based on the embedding of formal methods into a generic formal framework (here, the theorem prover Isabelle).

2 Semantic Aspects of Tool Integration

In short, Isabelle [5] is a *generic tactical LCF theorem prover*. This in particular means that the prover is implemented in the functional programming language Standard ML, centred around an abstract data type *theorems*, whose objects can only be constructed by applying the basic logical rules. The overall correctness of all formal activities is hence reduced to the correctness of this (small and well-investigated) logical engine. In our setting, Isabelle is used to construct proof tools based on logical embeddings of formal methods, and for transformational program development.

* This work has been supported by the German Ministry for Education and Research (BMBF) as part of the project UniForM under grant No. FKZ 01 IS 521 B2.

Without going into the details here, a *logical embedding* is a representation of a formal method inside Isabelle's logic. Two main logical embeddings have been developed in the course of the UniForM project, for the process calculus CSP [7] and for the specification language Z [4]. Although developing a logical embedding for real-life formal methods such as these is by no means a routine task, we still believe it is worthwhile the effort, for the benefit of being able to use it as much as for the benefit of validating the formal method itself– during the development of the formal embedding of CSP, a small but persistent error in the theory of CSP was found which went undetected for years.

The *Transformation Application System TAS* is based on Isabelle supporting transformational program development. It is generic in the transformations as well as in the formal method. Hence, we can for example instantiate it with transformations based on a simple fold/unfold paradigm in the theory of lists, obtaining what is essentially the Bird-Meertens calculus for functional programming, or with transformations based on refining CSP specifications.

Further, a generic graphical user interface for Isabelle and applications based on it has been developed making use of Standard ML's powerful abstraction and modularisation concepts. This way, to obtain a graphical user interface for any tool based on Isabelle, we merely need to instantiate the parameterised module (*functor*) with different parameters.

In conclusion, by developing a logical embedding for a given formal method we obtain a proof tool with a graphical user interface. If we can define a notion of transformation in this formal method (based on equality, refinement, or any other transitive-reflexive relation) we furthermore obtain a tool for transformational program development.

3 Technical Aspects of Tool Integration

The design of the UniForM-Workbench [3] supports the various technical issues of the tool integration process, such as *data integration* (sharing and exchanging data between tools), *control integration* (communication and inter-operation among tools and between tools and the integration framework), *presentation integration* (tool appearance and user interaction, i.e. look-and-feel), and others such as process and platform integration which we will not consider in this paper.

An integrated workbench for development of Z specifications (the *Z-Workbench*) is pictured in Fig. 1. The logical embedding of Z into Isabelle, called *HOL-Z*, has been started up in order to develop version `BB[3]` of the birthday book theory. The session has been invoked from within the development graph in the lower right corner by creating a new session `BirthDayBook[4]` from the session named `BirthDayBook[2]`. The corresponding configuration graph for the birthday book theory is shown in the upper right corner. This configuration imports version `Z[5]` of the standard Z library, the version graph of which is visible to its left.

Using the Workbench, SDE's are constructed by *encapsulating* existing development tools such as editors, model checkers and proof tools. The pure functional language Haskell [1,6], extended with a higher order approach to concurrency [2] is used as a central integration language, i.e. as GlueWare in this context. Integration of a tools such as Isabelle into this integration framework proceeds in three stages. First, the persistent objects types and their interrelationships are defined, and provisions are made for version and configuration management of these objects (data integration). Second, a Haskell interface is wrapped around the tool so that it can run under complete control of the Workbench (control integration).

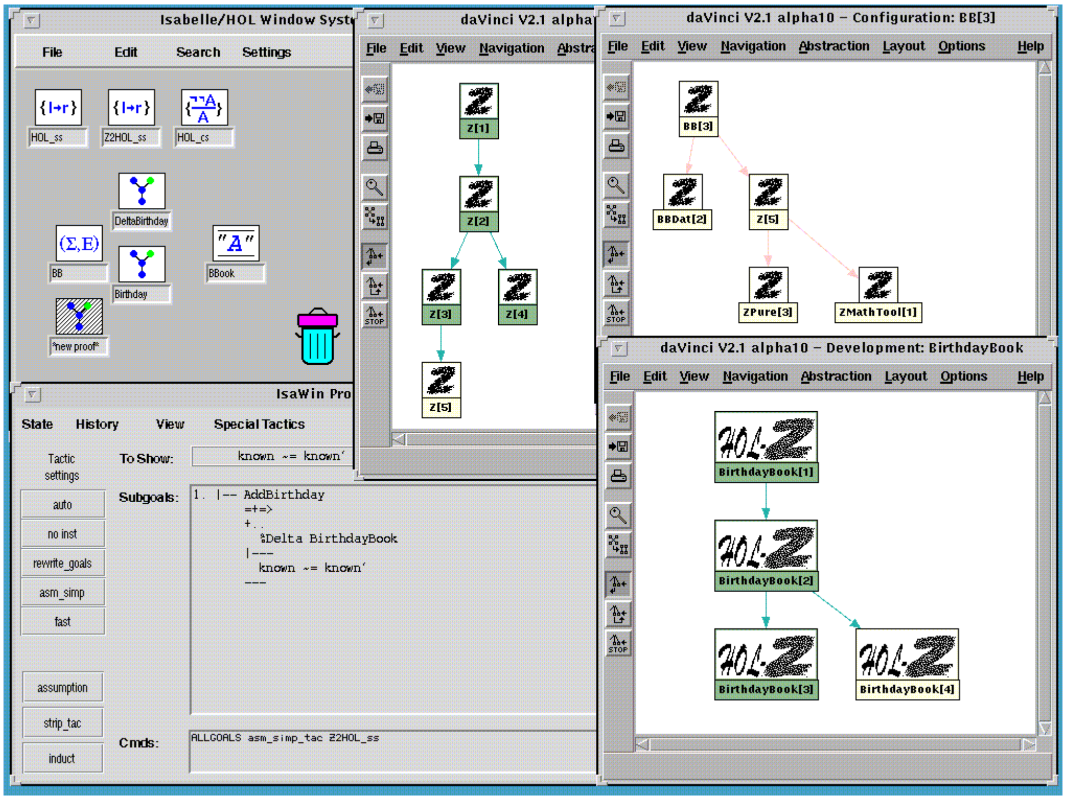

Fig. 1. Z-Workbench

In the third and last step, the user interfaces are designed and implemented such as the version and configuration graphs in Fig. 1 (presentation integration). All interaction with the system, such as starting up a tool or querying the content of the database, goes through these graphical views.

The integrated SDE offers a number of interesting advantages over a loosely coupled architecture working with a plain file system. All objects of the SDE, such as specifications and proofs are versioned and permanently kept consistent. Unlike a file system, the fine-grained dependency relationships between the objects, such as for example the proofs associated with a given theory, is maintained. The views provided by the integrated environment are furthermore at all times kept consistent with the current state of the repository — and all of this in a distributed multi-user setting.

The cost of getting these advantages is not high. The integration can be expressed at a very high level of abstraction, since we are using a powerful functional language complemented with an approach to concurrency (and event handling) akin to process algebras. The Workbench has furthermore been designed with abstraction and genericity in mind and offers an extensive and generic toolkit for supporting various aspects of data, control and presentation integration. One of these generic features of the system, to be discussed next, is a data model for version, configuration and session management which has been geared to handling Isabelle tools.

4 Generic Data Model

In the following, we give a generic data model which is suitable for any instantiation of the UniForM-Workbench centred around a logical embedding. The data model is given in Fig. 2 as an extended Entity-Relationship diagram, which is then systematically converted into the actual Haskell modelling. The two most basic object types are *folders* and *versioned objects*. Folders are the basic structuring mechanism within the repository, comparable to directories in conventional file systems. Versioned objects are the basic building blocks of the application, coming in four subtypes: tools, theories, sessions and proof scripts.

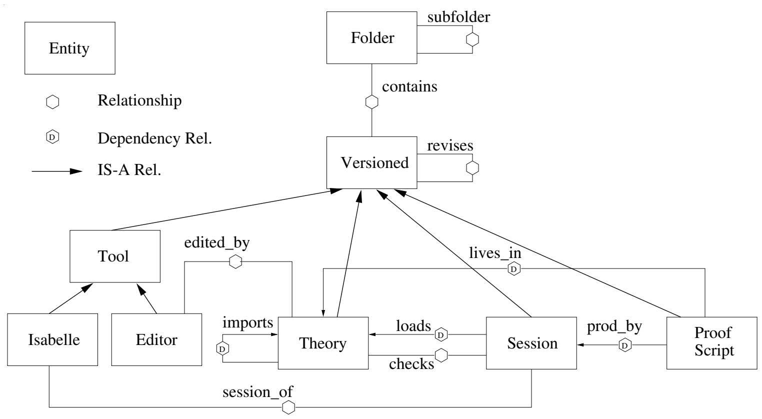

Fig. 2. UniForM-Workbench Generic Data Model

Tools are either those based on Isabelle, such as logical embeddings or TAS, or others such as a text editor. Since Isabelle itself offers a bewildering abundance of objects (i.e. types) and relations (i.e. functions) between them, finding an appropriate level of abstraction in the data model which is on the one hand fine enough to be able to benefit from the advantages of data integration, but on the other hand abstract enough to be still practical is the crucial task when integrating Isabelle into the Workbench. We have decided to distinguish between theories, sessions and proof scripts. *Theories* represent specifications, type and datatype definitions and so forth. They are structured hierarchically, and hence come with an *import* relationship, and are *edited_by* the text editor (i.e. not within Isabelle itself). *Sessions* represent the persistent state of a session with Isabelle. When a session is resumed, the proof work being done up to that point is reconstructed. *Proof scripts* are self-contained tactical Isabelle scripts, which are represented textually. Every proof script lives in the context of a theory, given by the *lives_in* link, and is produced by one Isabelle session given by the *prod_by* link. Each of theories, sessions, proof scripts comes in a different subtype for each of the Isabelle-based tools; e.g. the proof scripts for TAS correspond to transformational developments.

Further objects such as the documentation generated from a proof script within a session, or program code generated from executable specifications have for simplicity been omitted from Fig. 2.

5 Conclusion

We have sketched the approach to formal method tool integration used in the UniForM-Workbench. Logical embeddings based on the theorem prover Isabelle are used to handle semantic issues, and a loosely coupled architecture based on the pure functional language Haskell, extended with a higher order approach to concurrency, as GlueWare is used to handle the technical issues of tool integration. The resulting prototype gives an impression of the power of the modular, generic and functional technology employed.

The modular aspect allows the separate development of the components of the Workbench by different groups. Moreover, all the components (in particular, the logical embeddings and TAS) can be used on their own, and the GlueWare of the Workbench can be used for completely different tools, not necessarily connected to formal methods at all. But it is their combination that scales up a theorem proving environment which is correctness oriented and enables versioning (hence reproducibility) and overall semantic integrity (this piece of code or this generated proof obligation belongs to this specific theory) even in a distributed multi-user setting.

The decision to use strongly typed functional programming languages for both the Workbench and the tools based on Isabelle was crucial. Only by the abstraction and structuring mechanisms offered by these languages the integration could proceed at a very high level of abstraction. Working with tool support for formal methods this has become a crucial aspect, since the technology allows us to experiment with new ideas without being slowed down by a large implementation and maintenance overhead.

References

1. P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell - a non strict purely functional language, version 1.2. *ACM SIGPLAN notices*, 27(5):1–162, 1992.
2. E. W. Karlsen. The UniForM Concurrency Toolkit and its Extensions to Concurrent Haskell. In *GWFP'97: Glasgow Workshop on Functional Programming*, September 1997.
3. E. W. Karlsen. The UniForM WorkBench - a Higher Order Tool Integration Framework. Technical report, FB 3, Universität Bremen, 1998.
4. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, number 1125 in Lecture Notes in Computer Science, pages 283 –298. Springer Verlag, 1996.
5. L. C. Paulson. *Isabelle - A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer Verlag, 1994.
6. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages '96 (POPL'96), Florida*, 1996.
7. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 318– 337. Springer Verlag, 1997.