HOL-Z in the UniForM-Workbench - a Case Study in Tool Integration for Z *

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.

We demonstrate how to integrate several tools for the Z specification language into the Workbench, obtaining an instantiation of the Workbench suited as a software development environment for Z. In the core of the setting, we use the encoding HOL-Z of Z into Isabelle as semantic foundation and for formal reasoning with Z specifications. In addition to this, external tools like editors and small utilities are integrated, showing the integration of both self-developed and externally developed tools.

The resulting prototype demonstrates the viability of our approach to combine public domain tools into a generic software development environment using a strongly typed functional language.

1 Introduction

The need for tool integration has been widely recognised throughout software engineering. There is no single tool for all purposes. Moreover, we live in a highly distributed software production culture. Hence, it is likely to be impractical and unproductive to prescribe *ex cathedra* the particular tools to be used in a given development. Rather, it seems more advantageous to let software engineering teams employ the tools they are most comfortable with, and combine the various tools into one integrated *Software Development Environment* (SDE).

In response to this need, a number of tool-integration techniques have been developed. In the most simple approach, tools run independently, with the file system as a persistent store, and communication achieved by string-based "glueing" using Tcl [22], Expect [19] or Emacs Lisp. Unfortunately, this approach does not scale up to more sophisticated development environments, where features such as type-safe communication, persistent and

* 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.

distributed storage, version management and workflow management are required. In particular, this is the case for formal methods, where the semantic integrity of the documents produced by the various formal method tools have to be maintained, in order to enable a fine-grained configuration management of individual theorems, lemmata, proof obligations and proofs.

During the last decade, several attempts to meet this challenge were made based on environments for synthesising *tightly integrated* SDE's from the basis of abstract language specifications such as the Cornell Synthesizer Generator [27], Gandalf [8], PSG [1] or the Ipsen system [20]. These were not entirely satisfactory, due to either the development costs involved in requiring tools to be developed from scratch in a homogeneous language framework, or to the inapplicability of the language framework itself in the problem domain.

In the UniForM approach, tool integration is based on a *loosely coupled* architecture [18,28] where prefabricated tools are integrated on top of a *tool integration framework*. The UniForM Workbench, introduced in section 2, is the implementation of this framework, which offers support for data, control and presentation integration. It is generic, and we will here introduce its instantiation to the Z specification language (section 3), the Z Workbench. The semantic cornerstone of this instantiation is the encoding of Z into the theorem prover Isabelle, called HOL-Z. The encoding lends a semantic aspect to the integration, by providing type checking and the ultimate certification of the correctness of proof-scripts, allowing to maintain the semantic integrity of documents during the development process, and moreover allowing formal reasoning within Z specifications. HOL-Z and Isabelle will be the focus of section 4. This is followed by the main section (section 5) of our paper concerned with the data model of our Z Workbench. We demonstrate the Z Workbench at work with the canonical example at hand in section 6.

2 The UniForM-Workbench

The design of the UniForM-Workbench [13] reflects the guidelines of the ECMA Reference Model [4] (see Figure 1), which outlines the abstract functionality required to support the various dimensions of a tool integration process:

Data Integration addresses the issue of sharing and exchanging data between tools. It is mainly provided by the *repository manager*.

Control Integration is concerned with communication and inter-operation among tools and between tools and the integration framework. It is provided by the *subsystem interaction manager*.

Presentation Integration addresses the issue of tool appearance and user interaction, i.e. look-and-feel. It is provided by the *user interaction manager*.

Process Integration is concerned with the functions between tools of the environment and the end user, i.e. workflow management. It will be provided by the *development manager* (work in progress).

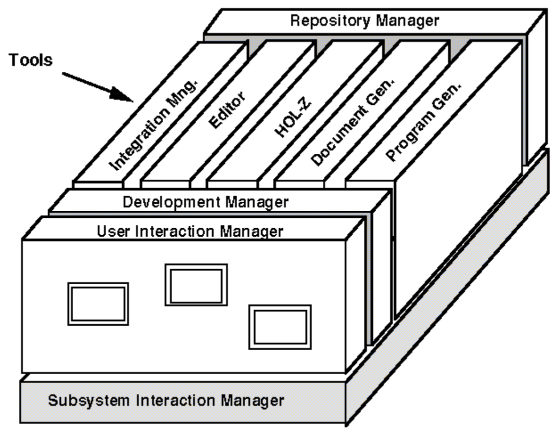

Fig. 1. ECMA Toaster Model

The ECMA reference model is also called the "ECMA Toaster Model", with the integration services in the rôle of the toaster. In this integration framework SDE's are constructed by *encapsulating* existing development tools such as editors, model checkers and proof tools. The pure functional language Haskell [9,24], extended with a higher order approach to concurrency [11] is used as a central integration language, i.e. as GlueWare[1] in this context. Each tool is encapsulated by wrapping Haskell interfaces around it using the integration manager. Section 5 will give an example of this process.

In the UniForM Workbench, an SDE is viewed as a reactive, event driven system. Events in such an environment amount to user interactions of the user interaction manager, change notifications of the active repository manager, operating system events and individual tool events. The subsystem interaction manager [10] takes the rôle of the central control component in this reactive systems architecture. It is structured as a network of communicating agents called *interactors*, whose behaviour is expressed using composable event values in the style of Concurrent ML [26]. New events can be defined from the basis of existing ones, using the *guarded choice* operator that provides a choice between two events, or the *event-action* combinator that combines an event with some additional reactive behaviour. This way, integration can be expressed at a very high level of abstraction.

The repository manager [31] provides databases services for the persistent storage of objects, and their exchange between tools. It is implemented by a Haskell encapsulation of the Portable Common Tool Environment (PCTE) [5] standard extended with version, configuration and workspace management. The repository manager is an *active database*: changes to an object

[1] Thank you to Phil Trinder for coming up with this wonderful term!

(i.e. committing a new version of an object) result in change notification events being sent to all other tools accessing the object. In order to integrate one or more tool, one first develops a data model in terms of the extended entity-relationship model underlying PCTE. From this semi-formal model, the actual implementation in terms of Haskell data types and type classes is derived in a systematic way. During the integration process, each tool is set up to work with persistent objects of the repository manager, rather than the plain files of the file system. Alternatively, the repository manager can export and import objects from the repository into the file system and back; this is less safe, but one does not need to change an existing tool's data access.

The user interaction manager [12] provides Haskell-Tk, a strongly typed, fully concurrent graphical user interface, with which graphical user interfaces for integrated tools can be constructed. Another component, the graph visualization system daVinci [7], is used to provide graphical user interfaces to the repository itself such as version and configuration graphs.

The Workbench obeys established industry standards such as the ECMA Reference Model, PCTE and Motif [6]. It has been implemented on the basis of public domain, off-the-shelf components supporting these standards. The repository manager for example is based on H-PCTE [3], whereas the user interaction manager integrates Tk [22] and the graph visualisation system daVinci [7]. Moreover, the Workbench offers a higher level of abstraction and uniformity than its underlying components, as well as additional utilities to support tool integration. The integration process is therefore much easier than if we had used these tools in their bare-bone form. One example is Haskell-Tk, which, as opposed to Tcl/Tk that it is based on, is strongly typed and fully concurrent. Another example is the repository manager, which provides version, configuration and workspace management on top of H-PCTE.

3 The Z-Workbench

The *Z-Workbench* is a Software Development Environment for Z built using the integration services provided by the UniForM Workbench — in other words, an instantiation of the generic framework provided by the Workbench to software development using Z. Its main component, providing a variety of services and the semantical underpinning of the integration, is the encoding of Z into Isabelle/HOL called HOL-Z [16].

HOL-Z reads Z specifications in the email format and does a type and consistency check (i.e. it will reject specifications which do not typecheck or are obviously inconsistent). One can then prove theorems within the specification, generate formatted documentation, or if the specification can be shown to be executable, generate program code. Thus, HOL-Z offers the following services:

- type and consistency check;
- symbolic theorem proving;

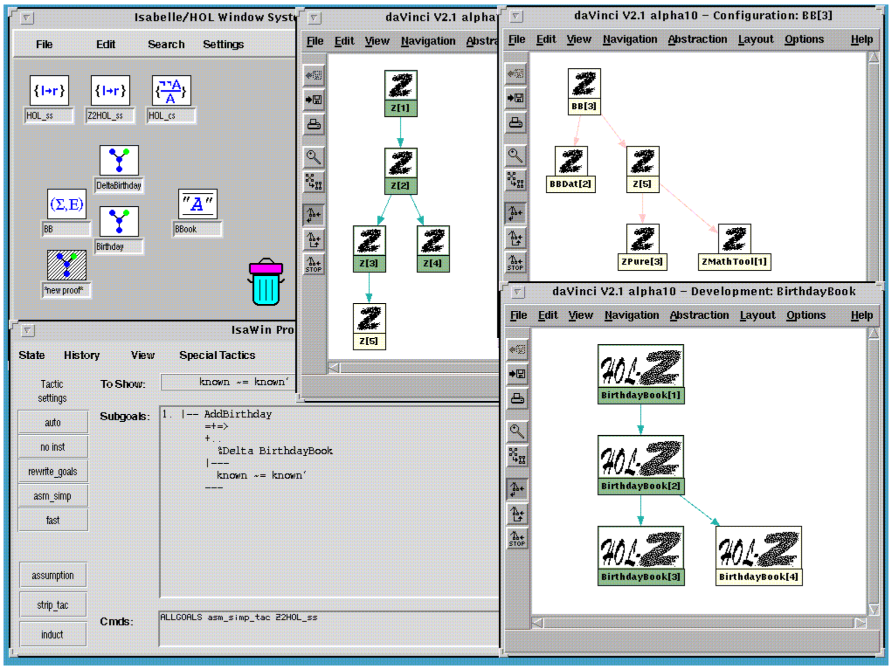

Fig. 2. Z-Workbench

– documentation generation;
– code generation.

Furthermore, to edit specifications a text editor is integrated. At present, this is a basic Unix text editor, but a structure-oriented schema editor could be integrated very easily as well. Finally, utilities such as the Unix tool `diff` are integrated, which is used by the version management.

A session with the Z-Workbench is pictured in Figure 2, where HOL-Z that 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 at the lower right corner by selecting the development named `BirthDayBook[2]` which yields a new session `BirthDayBook[4]`. 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, whose version graph is in turn shown to the left. The persistent objects taking part in the session are visualized in a brighter colour than objects that are not used by the session. These are also the objects that have been exported to the file system (behind the scene), in order to make them accessible to HOL-Z.

Integrating these tools into a Z-Workbench has numerous advantages over an implementation using stand alone tools working with a plain file system: all objects of the Z-Workbench, such as specifications and proofs are versioned and permanently kept consistent in a distributed, multi-user environment. Not only Z specifications, but also tools and proofs are put under control of the version management system. Old versions are never deleted, only outdated, and can always be reverted to. Hence, a formal development is always kept consistent — if a specification is outdated, the development using the old, outdated specification is still available.

The views provided by the Z-Workbench are at all times kept consistent with the current state of the repository. Changes to the repository are broadcasted to all running tools, such that changes made by one user can be recognised immediately by others. Developers may for example be notified immediately when a proof has been either proved correct or rejected, to avoid that development resources are wasted on redundant work or blind alleys.

Interaction with the environment is on a query by navigation basis, where the user browses through the object base using graphical user interfaces such as folder, version and configuration graphs visualised by daVinci. Tools are then invoked from within these views.

The Z-Workbench is, being based on the generic integration services of the UniForM Workbench, an *open integration framework* that can be extended with other development tools if needed. However, integration of other tools is not always as easy as could be hoped for, since it can be hampered by idiosyncrasies of prefabricated tools. In particular, the plethora of existing styles for Z specifications (email, LaTeX, box style) and deviations from these styles by existing tools may impose the need of converting a specification before it is passed to a tool. Still, the Workbench provides an excellent framework for hiding such technicalities, since the specifications are treated as logical objects, and the conversions will happen behind the scenes whenever possible.

4 Isabelle, HOL-Z and IsaWin

In short, Isabelle [23] is a *generic tactical LCF theorem prover*. Here, *generic* means that it is particularly suited for the encoding of different logics and formal methods, *tactical* means that it offers user-programmable proof support, and the *LCF design* means that the prover is centred around an abstract data type *theorems*, whose objects can only be constructed by applying the basic logical rules. The overall correctness on all formal activities is based on this (relatively small and well-investigated) logical engine.

As implementation, the prover can be viewed as a collection of ML types modelling theorems, proofs and theories, and ML functions modelling the possible proof activities.

The encoding of Z into Isabelle, called HOL-Z, benefited in particular from Isabelle's genericity, while the LCF design allowed the implementation of a versatile graphical user interface called IsaWin, not only for Isabelle itself, but also for other applications based on Isabelle, such as HOL-Z. We will now describe HOL-Z and IsaWin in greater detail. Their combination will yield a tool Win-Z for formal reasoning in Z specifications with a graphical user interface.

4.1 HOL-Z

HOL-Z is a shallow embedding which has the ability to preserve the structure of large specifications. This has the advantage that HOL-Z can be used for theorem proving in "real" Z-Specifications.

HOL-Z essentially consists of three parts:

- Isabelle Theories: An Isabelle theory (supporting the e-mail format of Z along the Z Standard [21]), a collection of extensions of the Isabelle/HOL-standard and the mathematical toolkit, the "library" of Z which consists itself of a collection of theories (relations, bags, sequences etc.)
- A loader for Z specifications
- A collection of tactics to support working with Z (prover)

HOL-Z supports the email format as proposed in the Draft Standard for Z. Thus, a schema is given in the e-mail format; a loader converts it into a semantic representation where the corresponding schema is a boolean-valued function.

The following schema known from the Birthday Book will be represented in the email input syntax as follows:

```
"+-- BirthdayBook ---
            birthday:(Name -|-> Date);
            known : Pow Name
|---
            known = dom(birthday)
 ---
"+-- AddBirthday  ---
        date_i:Date;
        name_i:Name;
        %Delta BirthdayBook
|---
        name_i~:known &
        birthday'= birthday Un {(name_i,date_i)}
 ---
"
```

The Z loader takes this schemas and produces a theorem in Isabelle which is pretty printed as shown below:

```
+-- BirthdayBook ---
    birthday : (Naturals -|-> Date); known : Pow Name
   |---
    known = dom(birthday)
   ---
```

4.2 IsaWin and Win-Z

IsaWin is a graphical user interface (GUI) for the theorem prover Isabelle. It represents the theorem prover's objects — theorems, proofs, theories, sets of rewriting rules — by icons. The operations on these objects making up the proof activity, are mostly affected by drag&drop, or to a lesser extent by activating buttons or menu entries. This gives the user access to nearly the full proof power of Isabelle without having to concern himself with ML and its syntax.

An interesting feature of the system architecture of IsaWin is its versatility. Since it is implemented entirely in Standard ML "on top" of Isabelle, it makes use of Standard ML's powerful abstraction and modularisation concepts, in particular its parameterised modules, called *functors*. This way, to obtain a graphical user interface for an encoding or other application based on Isabelle, we merely need to instantiate the functor with different parameters[2].

The instantiation of IsaWin for HOL-Z is called Win-Z. It contains *ZTheories* which are collections of schema declarations (sections) and serve essentially the same functions that Isabelle-theories do during theorem proving - forming the context a theorem is living in. Furthermore they incorporate an environment, *ZEnv*, containing an extralogical information about the occurrence of schemas, to respect the 'define before' principle of Z. In Figure 2, Win-Z can be seen on the left side.

5 The Integration

The development of integrated SDE's within the UniForM Workbench starts out with a number of prefabricated development tools that have not been integrated yet. In order to reach an integrated framework for Z development, issues regarding control, data and presentation integration must be addressed. Control integration means that the tool is given a Haskell API so that it can be controlled by the subsystem interaction manager, data integration that it is set up to interface the development objects maintained by the repository manager, and presentation integration that a graphical user interface along the guidelines of the Motif standard are wrapped around the tool, unless of course, the tool already comes with such an interface.

The resulting SDE is organised according to the *Model-View-Controller paradigm* [17], with the repository manager in the rôle of the *Model*, the user interaction manager in the rôle of the *View* and the subsystem interaction manager in the rôle of the *Controller* [14].

[2] In fact, IsaWin itself is the instantiation of a more generic graphical user interface called GenGUI [15].

5.1 Data Integration

The data model for the Z-Workbench is given in Figure 3 as an extended Entity-Relationship diagram, which is then systematically converted into the actual Haskell modelling (see Section 2). 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. Each folder may however be contained in a number of parent folders, and each folder may contain, in addition, a number of versioned objects. Versioned objects are the basic building blocks of the application. Each versioned object has two kinds of relationships: *dependency* and *development*. Dependency links model structural dependency, such as imports, and are given by all those links between the four subtypes of versioned objects marked Ⓓ in Figure 3. The development relationship defines *revisions* of an object, and can only exist between versioned objects of the same subtype. Dependency links are essential to model change propagation: if a new revision of an object is created, the revisions of the objects depending on this object will have to be updated as well. Thus, if we create a new theory by editing an existing one, all proof scripts living in the old theory will have to be updated as well in order to live in the new theory.

Versioned objects come in five subtypes: *tools*, *theories*, *sessions* and *proof scripts*.

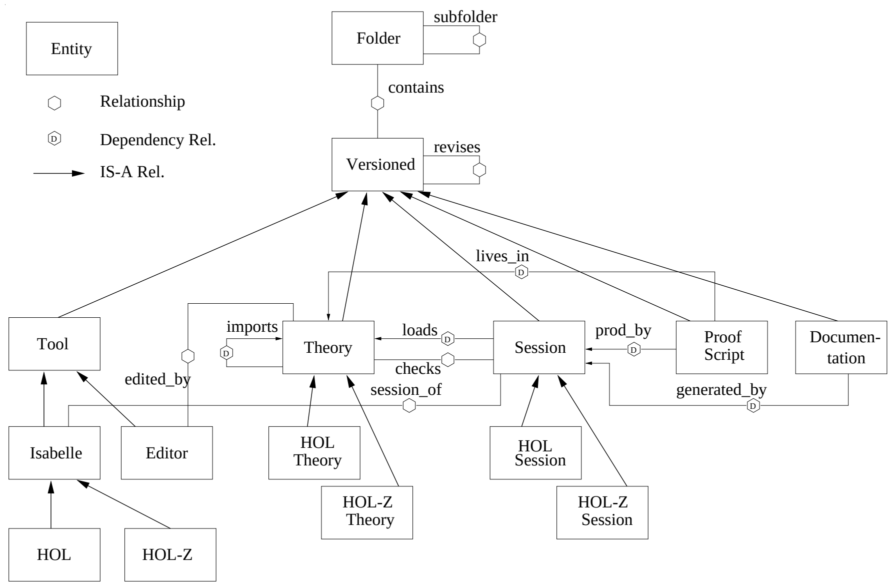

Fig. 3. Z-Workbench Data Model

Tools are either those based on Isabelle, such as HOL or the Z-encoding HOL-Z, or a text editor. By modelling tools as versioned objects, the Workbench is able to maintain and invoke different versions of the same tool. Tool revisions are used to represent new versions of the tool which are incompatible to existing developments; this is in particular relevant for Isabelle (and hence HOL-Z), where old tactical scripts will quite often not run on newer versions of Isabelle.

Theories represent specifications, type and datatype definitions and so forth. They come in subtypes corresponding to the different flavours of Isabelle; thus, there are HOL-Theories and Z-Theories, read by Isabelle/HOL and HOL-Z, respectively. Theories are edited externally (i.e. not within Isabelle itself). Theories are *checked*, if they have been successfully read by Isabelle (or HOL-Z), and become unchecked if they are subsequently edited. Theories in Isabelle are structured hierarchically, and hence come with an *import* relationship. Theories are *edited_by* the text editor.

Sessions represent the persistent state of a session with Isabelle or HOL-Z. Essentially this means that the user may save a session, and resume work later. When a session is resumed, the proof work being done up to that point is reconstructed. To maintain consistency of sessions, the distinction between checked and unchecked theories is important, since each session is reconstructed with the last checked version of the theory (which may be different from the last version of the theory, if it is unchecked). Every session is related (by the *is_session_of* link) to one specific HOL tool, which it runs on. Further, every session may load several theories (as indicated by the *loads* link), and if it successfully loads them makes them into checked theories (as indicated by the *checked* link).

Proof scripts are self-contained tactical Isabelle scripts. Since there is a multitude of objects and tactical operations within Isabelle making up these tactical scripts, modelling all of these in detail would be impractical, and we just treat them abstractly as a tactical script 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.

Finally, documentation can be generated out of a session having loaded several theories. These documentations can be subdivided into several modes (concerning theories, lists of theorems, proofs) and formats (e.g. HTML as done presently by Isabelle, LaTeX or RTF). The situation for generated documentation is analogous to generated *program code* (in the form of Standard ML) that can be generated from executable specifications. The program generator used for Z is currently under development.

Figure 3 is actually an abstraction from the more complicated modelling used in our integration. In particular, there are different types of sessions and proof scripts just as there are different kinds of theories and Isabelle tools; and the relationships between theories, sessions and proof scripts are on the level of these subtypes. There is however just one type of program code (*viz* Standard ML).

The Workbench uses Haskell classes to structure the code and to generalise the API's. The operations and events of relevance to the class of *versioned objects*, one of the central concepts of the repository, is for example outlined by the following class declaration:

```
class (RMIdObjectC vo) => RMVersionedObjectC vo where
   revise          :: vo -> IO vo
   getRevisions    :: vo -> IO [vo]
   getPredecessors :: vo -> IO [vo]
   revised         :: vo -> EV vo
```

The class provides (among others of course) a `revise` operation for creating new revisions, two computations for traversing the version tree (`getRevisions`, `getPredecessors`) together with a `revised` event that occurs whenever a versioned object has been revised. This class is then instantiated on need with theories, sessions and proof scripts.

5.2 Control Integration

All the tools of the Z-Workbench happen to be interactive Unix tools. Even HOL-Z can, although it comes with a graphical user interface, be squeezed into this category, since it is started and controlled from within a ML session.

The Workbench provides a utility called `Haskell-Expect` (inspired by `expect` [19]) for integrating such interactive Unix tools. It runs the Unix tool in the background, and lets the Workbench take over communication with the tool by simulating the user dialogue. The encapsulation of HOL-Z comes quite straightforward in this setting, as we shall demonstrate by the following computation that starts up HOL-Z loaded with a given session and theory:

```
startHolZ session theory = do {
  hz <- newExpect "holz" [];
  sync (match hz "^- ");
  sendCmd hz (load session theory);
  interactor (finished hz >>> do{sendCmd hz "quit();\n"; stop});
  return hz
  } where load s t = "load " ++ show s ++ " " ++ t ++ ";\n"
```

An `Expect` tool is created first that starts up HOL-Z as a ML session. We then wait until the ML interpreter responds with a prompt, as specified by the `match` event and responds by sending a command (`sendCmd`) that will start up WIN-Z and `load` the given `session` and `theory`. The command is forwarded in the form of a string as specified by the function `load`. An interactor is finally spawned off to catch the events that occur when WIN-Z finally returns control to the ML interpreter. The interactor responds by quitting the session with ML before it terminates itself by calling `stop`.

The Workbench is based on a higher order composable approach to event handling, where events are first class values. Base events can be combined into composite events using the guarded choice operator (`e1 +> e2`), and additional reactive behaviour can be glued onto existing events using the event-action combinator (`e >>> a`). The event that occurs when the session with HOL-Z is over, consists for example of two base events:

```
finished hz = (match hz "^- ") +> (match hz "uncaught.*\n")
```

The first event occurs when the session has terminated normally, i.e. `load` has finished and the ML interpreter has generated a prompt and awaits "user input". The second event should actually never occur, since it is generated when the session with HOL-Z has ended abruptly with an uncaught exception.

The Workbench can furthermore communicate with HOL-Z running as a server, in order to request services of HOL-Z or to inform HOL-Z about some external event of relevance to the session (new theorems etc.). The way this is achieved has already been demonstrated: `sendCmd` is used to submit commands to the tool and `match` for looking for specific response patterns.

5.3 Presentation Integration

The need for presentation integration within the Z-Workbench is quite restricted since all tools come with a graphical user interface. What remains is to develop the version and configuration graphs of the system, and provide the user with additional menus and user dialogues for calling and customising the services of the integrated Workbench.

When the user requests the system to open up a new view such as a version graph, an initial view is built first, i.e. the repository is traversed and the appropriate visualisation commands are passed to daVinci. A bunch of interactors are then associated with the graph to maintain consistency between the view and the underlying repository. An interactor for monitoring a single version looks like:

```
monitor g vo = interactor (
   revised vo >>>= (\rev -> do
       showRevision g vo rev
       monitor g rev
       redrawGraph g
       ))
```

The interactor reacts to the `revised` event by showing the new revision link within the current version graph. It then spawns off a monitor for the new revision `rev` and redraws the graph. It actually is as simple as this, although the interactor is in reality set up to listen to a couple of events more.

Generating views, and maintaining the consistency of views, is one issue, tool invocation is another. The services of the Workbench are invoked by using pull-down menus that are associated with the nodes within a view. For example, the version graph has a single interactor that listens to node selection events and menu invocation events:

```
controller g m o = interactor (
      nodeSelected g >>>= \o' -> become (controller g m o')
  +>  invoked m      >>>= \f -> f o
  )
```

The controller takes as parameter a handle to the current graph `g`, the application menu `m` and the selected object `o`. It responds to a node selection event by becoming an interactor that will apply commands to the new selection `o'` rather than the old one.

The second guard is concerned with handling menu invocations. The Workbench is higher order meaning that menus and menu items are functors that may return some value of interest to the application when the menu is invoked. In the case above, the application menu is set up to return the computation that defines the behaviour associated with the menu item. This computation is then applied to the object currently selected.

The missing link is the following piece of code that ensures that a new session with HOL-Z is started whenever the user clicks the `"Revise"` button associated with a HOL-Z session:

```
button [text "Revise", command (return newHolZSession)]

newHolZSession vo = do
    vo' <- revise vo
    (session,theory) <- exportSession vo'
    startHolZ session theory
```

A new revision is created first and all files of relevance to the HOL-Z session are then exported to the file system. A new development using HOL-Z is then started in the context of the given session and theory.

5.4 Experiences gained from the Prototype

The Workbench provides, being based on Haskell extended with a higher order approach to concurrency, a high level of abstraction and expressive power that comes very close to the one of constructive formal specifications. There are several key features in achieving this.

First of all we benefit from Haskell's expressive power by being a higher order and strongly typed language. Classes are furthermore used to structure the code and to standardise the interfaces to the system. The sequential behaviour of the system is expressed in terms of IO computations that have a theoretical foundation in the monadic approach to IO. A more practical consequence is that we have an extensible language framework that can be enhanced with new computational paradigms on need. The innovative part of the system however is the approach to event handling that treats all events of the system, whether they are user interactions, data base change notifications, operating system signals or individual tool events in terms of first class composable values that entirely hide the source of the event. Tool integration can therefore be expressed in a style that comes close to the one used, had we just made a formal specification using process algebras. The difference though is that Haskell is executable - and highly efficiently compiled as well.

6 Proving in the Birthday Book

In this section we will use the classical birthday book example to demonstrate the look-and-feel for Win-Z and its interaction with the surrounding Z-Workbench. This will be done with a proof of a tiny property of *AddBirthday*, namely that the set of names known to the system will be different with the addition of a given new name.

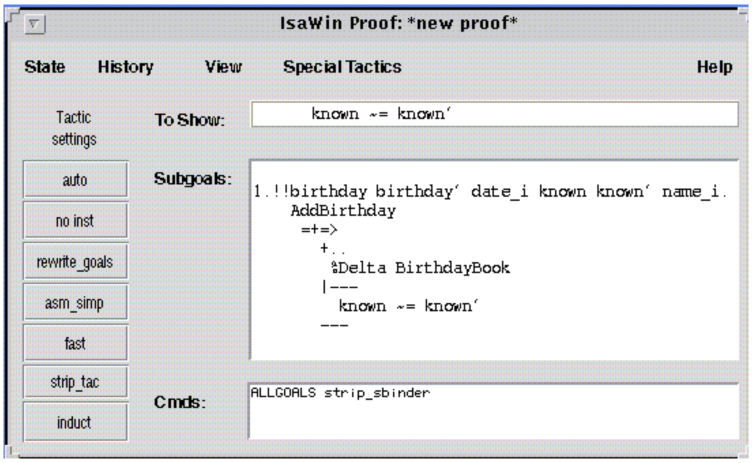

Fig. 4. Making implicit variables explicit

First of all, we defined a Z-theory *BB.zthy*, which is a Z paragraph containing the Z schemas defining the Birthday Book. The leading fragment of this paragraph has already been shown in section 4.

We state our desired property as follows:

```
|-- AddBirthday =+=> +.. %Delta BirthdayBook |--- known ~= known' ---
```

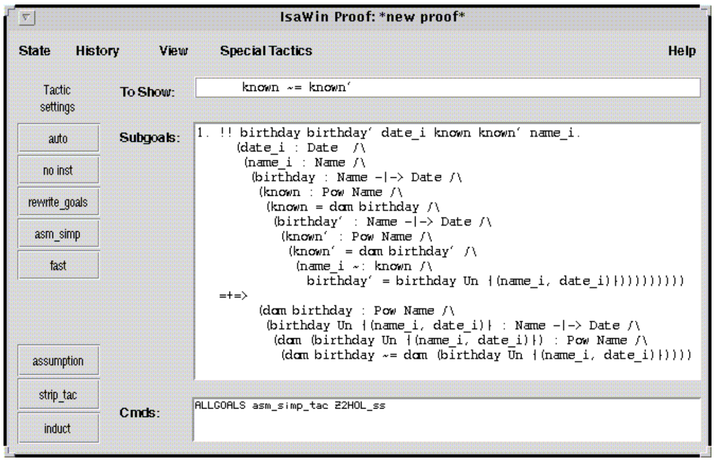

Fig. 5. After Schema Expansion and Simplification

We start with a Z-session where *BB.zthy* has already been loaded, and enter this goal with Win-Z. Figure 4 shows a screenshot where the goal above has been refined with one tactical step, that makes all the implicit free variables (the parameters of the schema), explicit by universally quantified variables.

We proceed now by expanding the schema definitions for *AddBirthday* and *BirthdayBook*. Further, elementary simplification leads to the following proof-state as shown in Figure 5. At this level, our fictive user interrupts the development, and wishes to save the session. Figure 8 shows this possibility, where the Workbench saves the HOL-Z session.

Another user goes to the version graph of the Z-session shown in section 3. The Workbench has meanwhile reacted to the change notifications of the repository manager, and has updated the version graph accordingly to show the new revision. The user just clicks on this generated version, and the Z-Workbench will start Win-Z in exactly the state saved by our first user.

After applying some minor tactics the goal is divided now in small subgoals as shown in Figure 6. The last three subgoals are proven automatically by using Isabelle's decision procedures for sets. The first one involves some knowledge about domains. One has therefore to use the simplifier sets related to the Z Mathematical toolkit. Figure 7 shows the result of this simplification.

At this level of the development the user needs a lemma about subsets and union, stating that if B is not included in A, then A is not equal to the union of A and B:

```
not(B <= A) ==> A ~= A Un B
```

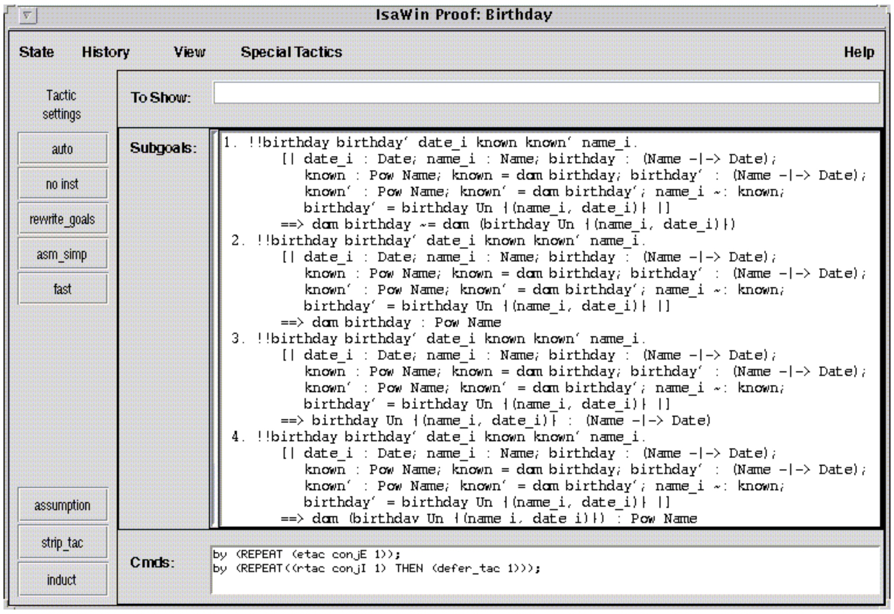

Fig. 6. Breaking up the Conjunction of the Conclusion

Now, using this lemma produces the following proof state as shown in Figure 8, Isabelle's decision procedure will do the rest. Again, our new user can save the newly generated session and extract from it the proof-script underlying the demonstrated proof-development.

7 Conclusion

We have seen an instantiation of the UniForM-Workbench for Z based on a specific LCF-style prover environment called HOL-Z[16]. The resulting prototype gives an impression of the power of the modular, generic and functional technology employed.

The modular aspect allows that the components of the Workbench can be developed by different groups of developers and users. It is perfectly possible to use the pure encoding HOL-Z or its combination with Win-Z in itself. It is possible to use the GlueWare of the Workbench for completely different tools, not necessarily connected to Formal Methods at all.[3] But it is the combination of these three that scales up a correctness-oriented, but notoriously difficult to use prover-environment to an formal methods environment, that enables versioning (hence reproducibility) and overall semantic integrity (this piece

[3] This has been done for the Hugs-Workbench [10,14].

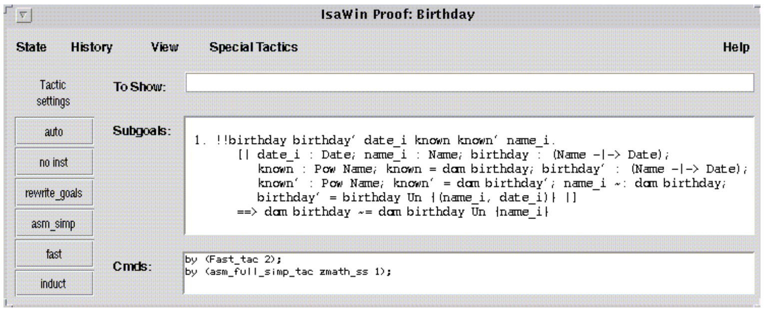

Fig. 7. Simplification with the Mathematical Toolkit

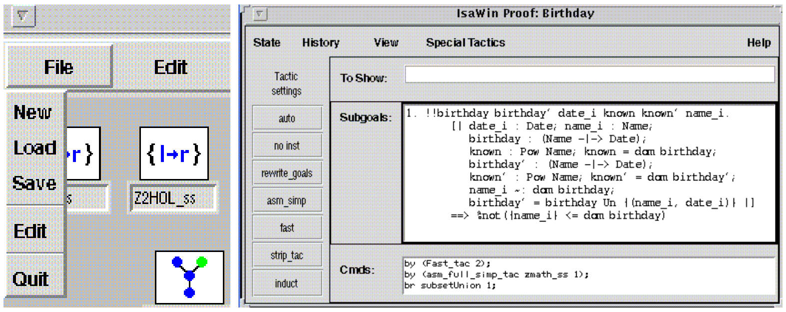

Fig. 8. HOL-Z Session (left), Proof State after Lemma Introduction (right)

of code or this generated documentation belongs to this specific state of a Z-theory) even in a multi-user setting. Thus, it is possible to reconfigure the Workbench with little effort with new external tools on the one hand and new semantic embeddings like [30].

The pervasive generic aspect of our technology stresses this toolkit-character of the Workbench even more. The tools themselves are built by many components, for example the generic graphical user-interface GenGUI, that are designed to be independent from each other and that can be *instantiated* for there particular application: In principle, GenGUI could be used for a completely different, sml-based LCF-prover such as the HOL/HOL System.

We are very happy with our decision to use strongly-typed, state-of-the-art functional programming languages for both the Workbench and Win-Z. The combination of a purely functional language extended with a higher order model to concurrency, allows the integration to be done 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.

7.1 Related Work

A short discussion of related *general* tool-integration techniques has already been done in section 2. In this section, we will therefore concentrate on integration techniques used for formal method tools.

Roughly, we divide existing approaches into two categories; first, ad-hoc approaches, which we refer as "Glue&Pray", and secondly more refined ones labelled "Glue&Play". This division is clearly debatable, and the two categories are rather the extreme poles in a continuum, and a tool belonging to the former, admittedly polemically labelled class is not necessarily a bad tool. But we believe that formal method tools as particularly complicated software products suffer to a larger extent from poor implementation technology than other software, be it with respect to the reliability of their results or be it with respect to maintainability necessary to cope with the rapidly evolving field. A case in point being PVS, which although as a prover is as powerful as Isabelle, and more widely used, suffers from an untyped implementation and interface language; or Tcl-based interfaces to LCF-provers like TkHOL or XIsabelle, which rarely survive the rapidly changing revisions of the underlying prover.

Examples for the "Glue&Play" approach are the Cogito [2] system or KIV [25]. As tools, these are clearly superior to our Workbench in terms of availability, user-friendliness and stability; but again, we believe they suffer from a system architecture which does not have a theorem prover as powerful and versatile as Isabelle at its heart, and which does not allow them to keep up with changes as good as our Workbench, the modular design of which allows easy exchange of parts which are outdated or superseded by newer developments.

7.2 Future Work

With respect to the Workbench for Z, more tools supporting different documentation formats like LaTeX, RTF or HTML on the one hand and animators on the other would be desirable extensions to the existing prototype. The Win-Z component needs the integration of the code-generator currently under development and more GUI-specific tactical support.

Another line of extension is the integration of other Z-Tools (like animators or test-case generation tools) *not* based on the Isabelle-Kernel. This kind of integration requires, due to the variety of syntactical constructs of Z, very carefully constructed translators, which often represent a reliability problem to the integration in practice. Paradoxically, it seems easier for us to integrate other Isabelle-Encodings (like our CSP-Encoding) plus formally proven sound combination formalisms for Z and CSP (such as [29]) into our logical core engine, turning the Workbench into a technical framework for combined reasoning over them.

In our actual prototype, the granularity of data to be exchanged between different sessions is still rather coarse. For instance, the Workbench has no access to the components of proof scripts. A refinement of the data model could allow individual proofs to be transferred from one session to another. This could be substantially enhanced if general merge techniques on proof scripts (as developed for a specialised logic inside KIV) were available. All these techniques could be combined with an active *change propagation* mode of the Workbench — i.e. the Workbench starts the logical engine in a batch mode, passes it a textually modified Z-theory, causes a re-evaluation of depending proof scripts with the aim to recertify as much as possible, and to save the resulting state of the logical engine in a session that is ready for further interactive development.

References

1. R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, October 1986.
2. A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor. Cogito: A Methodology and System for Formal Development. *International Journal of Software Engineering*, 4(3), 1995.
3. The H-PCTE Crew. H-PCTE vs. PCTE, version 2.8. Technical report, Universität Siegen, June 1996.
4. ECMA. Reference Model for Frameworks of Software Engineering Environments. Technical Report TR/55, European Computer Manufacturers Association, June 1993.
5. ECMA. *Portable Common Tool Environment (PCTE) — Abstract Specification*. European Computer Manufacturers Association, 3 edition, December 1994. Standard ECMA-149.
6. Open Software Foundation. *OSF/Motif Series*. Prentice Hall, 1992.
7. M. Fröhlich and M. Werner. daVinci V2.0.3 Online Documentation. Universität Bremen, `http://www.informatik.uni-bremen.de/~davinci`, 1997.
8. A.N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, December 1985.
9. 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.
10. E. W. Karlsen. Integrating Interactive Tools using Concurrent Haskell and Synchronous Events. In *CLaPF'97: 2nd Latin-American Conference on Functional Programming*, September 1997.
11. E. W. Karlsen. The UniForM Concurrency Toolkit and its Extensions to Concurrent Haskell. In *GWFP'97: Glasgow Workshop on Functional Programming*, September 1997.
12. E. W. Karlsen. The UniForM User Interaction Manager. Technical report, FB 3, Universität Bremen, 1998.
13. E. W. Karlsen. The UniForM WorkBench - a Higher Order Tool Integration Framework. Technical report, FB 3, Universität Bremen, 1998.
14. E. W. Karlsen and S. Westmeier. Using Concurrent Haskell to Develop Views over an Active Repository. In *IFL'97: Implementation of Functional Languages*, September 1997.

15. Kolyang, C. Lüth, T. Meier, and B. Wolff. TAS and IsaWin: Generic interfaces for transformational program development and theorem proving. In M. Bidoit and M. Dauchet, editors, *TAPSOFT 97': Theory and Practice of Software Development*, number 1214 in Lecture Notes in Computer Science, pages 855–859, Lille, France, April 1997. Lecture Notes in Computer Science.
16. 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.
17. G. Krasner and S. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
18. M. Lacroix and M. Vanhoedenaghe, editors. *Tool Integration in an Open Environment*, volume 387 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.
19. D. Libes. expect: Scripts for Controlling Interactive Processes. In *Computing Systems, Vol 4, No. 2*, Spring 1991.
20. Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*. Springer, 1996.
21. J. Nicholls and The members of the Z Standards Panel. Z notation, September 1995. Available at `http://www.comlab.ox.ac.uk/oucl/groups/zstandards/`.
22. J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.
23. L. C. Paulson. *Isabelle - A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer Verlag, 1994.
24. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages '96 (POPL'96), Florida*, 1996.
25. W. Reif, G. Schellhorn, and K. Stenzel. Proving system correctness with KIV. In M. Bidoit and M. Dauchet, editors, *TAPSOFT 97': Theory and Practice of Software Development*, number 1214 in Lecture Notes in Computer Science, pages 859– 862, Lille, France, April 1997. Lecture Notes in Computer Science.
26. J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.
27. T. Reps. *Generating Language Based Environments*. PhD Thesis, Cornell University. MIT Press, 1983.
28. D. Schefström and G. van den Broek. *Tool Integration*. Wiley, 1993.
29. G. Smith. A semantic integration of object-Z and CSP for the specification of concurrent systems. 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 62– 81. Springer Verlag, 1997.
30. 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.
31. S. Westmeier. Verwaltung versionierter persistenter Objekte in der UniForM WorkBench (UniForM OMS Toolkit). Master thesis, FB 3, Universität Bremen, January 1998.