The PROSPER Toolkit *

Louise A. Dennis[1], Graham Collins[1], Michael Norrish[2], Richard Boulton[3], Konrad Slind[2], Graham Robinson[1], Mike Gordon[2], and Tom Melham[1]

[1] Department of Computing Science, University of Glasgow, G12 8QQ, UK
[2] Computer Laboratory, University of Cambridge, CB2 3QG, UK
[3] Division of Informatics, University of Edinburgh, EH1 1HN, UK

Abstract. The Prosper (Proof and Specification Assisted Design Environments) project advocates the use of toolkits which allow existing verification tools to be adapted to a more flexible format so that they may be treated as components. A system incorporating such tools becomes another component that can be embedded in an application.

This paper describes the Prosper *Toolkit* which enables this. The nature of communication between components is specified in a language-independent way. It is implemented in several common programming languages to allow a wide variety of tools to have access to the toolkit.

1 Introduction

Modern system design, both for hardware and software, must meet ever-increasing demands for dependable products of high quality, with shorter design times and early error detection. Incremental improvements to conventional design methods and tools are not enough. More powerful techniques of specification and analysis based on formal techniques are essential parts of new methods for design.

Formalisms provide specification and analysis at high levels of abstraction, so that designers can express and check a wider range of properties than with conventional techniques. This permits better structuring of complex systems, earlier detection of errors (leading to lower time to market), and higher quality.

Making effective use of formal techniques does *not* have to mean doing 'total verification' against 'complete formal specifications' or designing step-by-step with a formal refinement theory. This rather modernist Formal Methods programme has still to deliver significant benefits to large-scale design practice, and verification has, in consequence, remained largely an academic activity regarded sceptically by industry. Instead, one can view formal analysis (or 'property-checking') of systems as an advanced or more effective form of testing—whose objective is not necessarily to have a strong assurance of correctness, but rather to eliminate more bugs, earlier in the design process [10].

At present, a developer wishing to incorporate verification capabilities into a CAD or CASE tool, or any application, will face a difficult choice between

* Work funded by ESPRIT Framework IV Grant LTR 26241

creating a verification engine from scratch and adapting parts of one or more existing tools. Developing a verification engine from scratch is time-consuming and will usually involve re-implementing existing techniques. Existing tools, on the other hand, tend not to be suitable as components that can be patched into other programs. Furthermore, a design tool should embed verification in a way that is natural to a user, i.e. as an extension to the design process (much like debugging is an extension to the coding process). The verification engine must be customised to the application.

The PROSPER project[1] is addressing this issue by researching and developing a toolkit that allows an expert to easily and flexibly assemble *proof engines* to provide embedded formal reasoning support inside applications. The ultimate goal is to make the reasoning and proof support invisible to the end-user—or at least, more realistically, to incorporate it securely within the interface and style of interaction they are already accustomed to.

This paper describes the PROSPER toolkit and the methodology of building systems with it. §2 gives a high level view of the toolkit and what a resulting system may look like. While it is inappropriate to give much technical detail here,[2] §3 tries to give a flavour of the specification for communication. §4 discusses the methodology for building systems with the toolkit. §5 presents a case study. §6 gives an overview of related work.

2 Design Tools with Custom Proof Engines

A central part of PROSPER's vision is the idea of a proof engine—a custom built verification engine which can be operated by another program through an Application Programming Interface (API). A proof engine can be built by a system developer using the toolkit provided by the project. A proof engine is based upon the functionality of a theorem prover with additional capabilities provided by 'plugins' formed from existing, off-the-shelf, tools. The toolkit includes a set of libraries based on a language-independent specification for communication between components of a final system. The theorem prover's command language is treated as a kind of scripting or glue language for managing plugin components and orchestrating the proofs.

The central component is based on a theorem prover because this comes with ready made concepts of term, theorem, and goal, which are important for managing verifications. A side benefit is that all the functionality in the theorem prover (libraries of procedures, tactics, logical theories, etc.) becomes available to a developer for inclusion in their custom proof engine. This does not prevent theorem proving being very lightweight if desired.

A toolkit has been implemented based around HOL98, a modern descendent of the HOL theorem prover [11]. HOL98 is highly modular which suits the PROSPER approach of building up a proof engine from a set of components (be

[1] `http://www.dcs.gla.ac.uk/prosper`

[2] Technical documentation is available from the authors.

they HOL libraries or external plugins). It also contains a number of sophisticated automatic proof procedures. HOL's command language is ML [19] (a strict functional programming language) extended with HOL's own theorem proving functions which extend ML to include the language of higher order logic [6]. This allows a developer to have a full programming language available in which to develop custom verification procedures. Proof procedures programmed in the proof engine are offered to client applications in an API. This API can be accessed as a verification library in another programming language.

The toolkit provides several plugin components based on external tools which offer APIs to a proof engine. It also provides support to enable developers of other verification tools to offer them as PROSPER plugins.

The application, proof engine and plugins act as separate components in the final system (Figure 1). In the first prototype they are also separate processes.

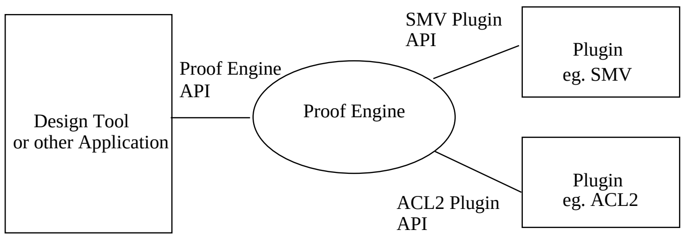

Fig. 1. A system built with the PROSPER toolkit

Communication between them is treated in a uniform manner specified by the PROSPER *Integration Interface*.

Work is currently underway to use this technology to add verification capabilities to IFAD's VDM-SL Toolbox [8]. The project is also building a *Hardware Verification Workbench*. This will allow specifications in Verilog and VHDL to be checked by a proof engine that incorporates a model checker.

Example 1. The IFAD VDM-SL Toolbox is a software design tool supporting the specification language, VDM-SL.

The proposed extensions to the Toolbox centre around the discharge of proof obligations generated by type invariants. Invariants are undecidable, so the automatic type checking functionality of IFAD's toolbox does not check their truth.

Many invariants can be discharged by first order logic decision procedures. To utilise these, the invariant condition needs to be translated from VDM-SL into first order logic. In particular, any user-defined functions must be simplified away. More complex simplification and theorem proving techniques can be used when the conditions fall outside the domain of first order logic. If an automatic proof attempt fails then a user must be able to intervene and guide a proof by hand.

This analysis suggests that the VDM-SL Toolbox requires a first order logic plugin; a proof engine with an embedding of the semantics of VDM-SL in higher order logic, specialised procedures for simplifying and translating VDM-SL expressions into first order logic (a subset of higher order logic) and some more complex proof techniques; procedures for the automatic generation of invariant proof obligations in the Toolbox itself, and a Toolbox specific interface to the proof guidance facilities provided by HOL. These elements can all be constructed together into the IFAD Toolbox using the PROSPER toolkit.

3 The Prosper Integration Interface

A major part of our methodology is the PROSPER Integration Interface (PII), a language-independent specification of communication for verification. This specification is currently implemented in several languages (C, ML, Java and Python) allowing components written in these languages to be used together.

The PII consists of several parts. The first is a datatype, called *interface data*, for all data transferred between an application and a proof engine and between a proof engine and its plugins. A major part of the datatype is the language of higher order logic used by HOL and so any formula expressible in higher order logic can be passed between components. Many plugins operate with logical data that is either already a subset of higher order logic (e.g. predicate calculus and propositional logic) or embeddable in it (e.g. CTL). The second part consists of a datatype for the results of remote function calls and support for installing and calling procedures in an API. There are also parts for managing low level communication, which are largely invisible to an application developer.

The PII distinguishes between clients and servers. An application is a client of a proof engine which is a client of any plugins it may have. Any server built using the toolkit offers an API to clients. This API describes its functionality in terms of interface data and a result type (which signals whether a function succeeded or failed and returns interface data). As far as an application or verification tool developer is concerned, all components talk the language of these datatypes; The details of translating calls made between components into and out of the raw communication format are entirely invisible.

The PII can be viewed as consisting of the layers in Figure 2. The lower layers deal with communication (handling interrupts and so on). The translation layer takes the raw data passed along the communication channel and translates it into the language's implementation of interface data. The application support layer supplies functions for working with interface data, starting and managing communication between components, and support for working with the API. On top of this sits the target application, proof engine or plugin. The application support layer is the highest specified by the PII.

3.1 Interface Data

Interface data is the high level language passed between components of the system. It can be used to represent a large number of types and expressions.

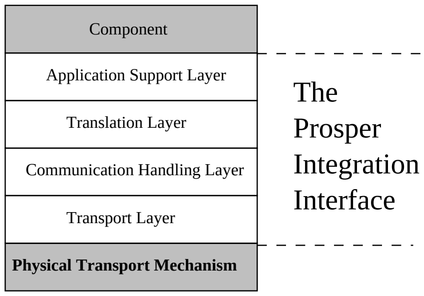

Fig. 2. The layers of the PII

The PII gives an abstract specification for interface data, but the exact form of the operations and their usage depends on the implementation language.

Each element of interface data has three operations, a constructor, a destructor, and a query (is_a) function which can be used to establish how an expression has been constructed.

Interface data consists of elements based on several standard types (booleans, integers, strings, etc.) and lists (allowing tree structures). It also contains special elements for handling proof concepts (logical terms, the types of logical terms and theorems).

Logical Terms and Types. Logical terms are central to the Prosper toolkit and are not a standard feature of any programming language. Logical terms are based on the syntax of classical higher order logic [6] and are the basic expressions for communicating logical information (e.g. conjectures that need proving). As usual with Church-style formulation, there are four basic elements for variables, constants, function applications and lambda abstractions (with associated constructor, destructor and query operations). Higher order logic is typed so it is also possible to query a term for its logical type.

These four elements are too low level for everyday use. This is reflected in HOL which supplies *derived syntax* to provide a usable set of constructors for common term structures. This approach is also adopted by the interface data specification. The interface data derived syntax consists of the usual logical connectives and quantifiers plus a few other common constructs.

3.2 API Support

The PII application support layer provides functions to allow client and server components to handle remote procedure calls. It uses a datatype, *interface data result*, with constructors `mk_succeeded:interface_data -> interface_data_result` and `mk_failed:interface_data -> interface_data_result` to report back the results of calls.

A client can use the operation, `call_server`, which calls a function in another component's API, referenced by a string, and supplies it with interface data as arguments. It returns an interface data result.

A server has some database manipulation functions for using an *API database* containing functions of type `interface_data -> interface_data_result` referenced by strings. These are used to process calls from a client.

3.3 Connection Support and Lower Layers

The PII application support layer includes client side functions for connecting to and disconnecting from servers. At present a server has to be started and stopped manually, externally to the client. In future we intend to allow servers to be started and stopped by the client.

The low level details of communication handling are only relevant to those wishing to implement the PII in various languages. The underlying communication is currently based on Internet sockets.

4 Using the Toolkit

The basic PROSPER toolkit consists of relatively little: a small subset of HOL, called the *Core Proof Engine*. This consists of a theorem type, inference rules for higher order logic and an ML implementation of the PII. The Core Proof Engine forms the core of all proof engines built using the PROSPER Toolkit. A developer can write extensions to the Core Proof Engine and place them in an API to form a custom API.

Many applications will require a version of the PII in an implementation language other than ML. The toolkit currently includes PII implementations in several languages and a couple of pre-made plugins (the SMV model checker [17] and Prover Technology's proof tool [23, 22]) which can be added into proof engines. Third party plugins are already also available for ACL2 [4][3] and Gandalf [25, 14].

Developing an application using the toolkit is, potentially, a large task involving several languages and programs. We have identified three aspects of working with the toolkit which separate out the tasks involved. These partition the effort into the areas most likely to be undertaken by distinct groups of people. The three aspects also help identify which parts of a final system should be responsible for which tasks.

4.1 The Theorem Prover Aspect

The theorem prover aspect (Figure 3) mainly involves ML programming. This programming effort focuses on developing custom procedures and placing them in an API which extends the Core Proof Engine. A developer will have access to

[3] Currently unpublished, but available at `http://www.cl.cam.ac.uk/users/ms204/`

the entire command language of the theorem prover and a set of entrypoints into as many plugins and theories as they might wish and are available. They will be able to develop custom verification procedures and theories within a strongly typed, LCF-style, environment. The outcome will be a custom proof engine with an API that can be passed on to the developer of an application as a verification library.

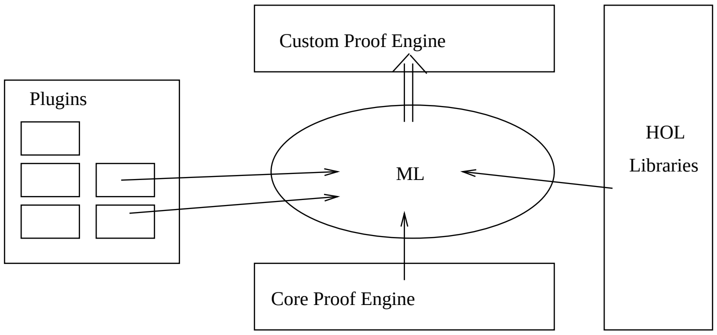

Fig. 3. The Theorem Prover Aspect

On the logical side, a custom proof engine will probably include an embedding of the formalism used by the application into higher order logic. It will also include automated proof procedures tailored for the application. These procedures may well use plugin decision procedures (e.g. for predicate or propositional logic) or even include, as plugins, verification tools previously developed as support for the application. Construction of such procedures may be a simple process of linking together highly-developed proof libraries and/or plugins or it may require more complex development.

Although implementations of the PII provide basic functions for calling proof engine APIs from clients, any serious application will want to wrap up the API with language-specific bindings (e.g. In an object oriented language it would be natural to present functions in a proof engine's API as methods in some verification class, thus hiding all instances of `call_server` from the application aspect developer). This can only be done if the implementation language of the target application is known.

Example 2. In hardware verification there exist many decision procedures for verifying designs. PROSPER sees its main application here as verification tasks that can be handled automatically through combinations of plugin decision procedures and theorem proving (see §6). These combined procedures will be developed in the theorem prover aspect and presented as the API to a custom proof engine.

4.2 The Application Aspect

The application aspect (Figure 4) focuses on the incorporation of a custom proof engine into an application so that it appears as a natural extension of the application's functionality. A developer will have access to an API offered by a proof engine already customised to their tool.

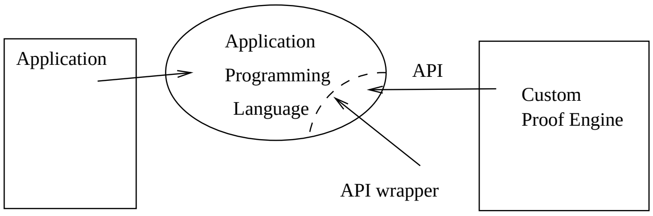

Fig. 4. The Application Aspect

The aim of PROSPER is that verification should fit as seamlessly as possible into the design flow. We envisage that most of the programming at this stage will focus on this task.

Example 3. The project is investigating the use of a natural language interface[13] to the Hardware Verification Workbench that will translate statements about circuits, in the normal technical language of engineers, into CTL propositions that a proof engine can verify. This will allow engineers to be largely unaware of the mathematical verification that is taking place.

4.3 The Plugin Aspect

The PROSPER toolkit supports a third aspect (Figure 5). The developer of a verification tool can adapt it so that it can be used as a PROSPER plugin. A plugin developer programs both in ML and in the plugin's own implementation language. The developer will place chosen entrypoints to the plugin into an API database. In the plugin's implementation language they will translate any arguments needed by these functions into interface data. In the theorem prover's command language they will need to unpackage these entrypoints again so they present themselves as language-specific bindings in that language (ML). In particular any additional theories required (i.e. an embedding of the logic used by the plugin into higher order logic) should be provided by the plugin developer. The plugin aspect is analogous to the theorem prover aspect except that it is assumed that the underlying functionality for the API is already implemented and provision of language-specific bindings is strongly recommended since the target language is known.

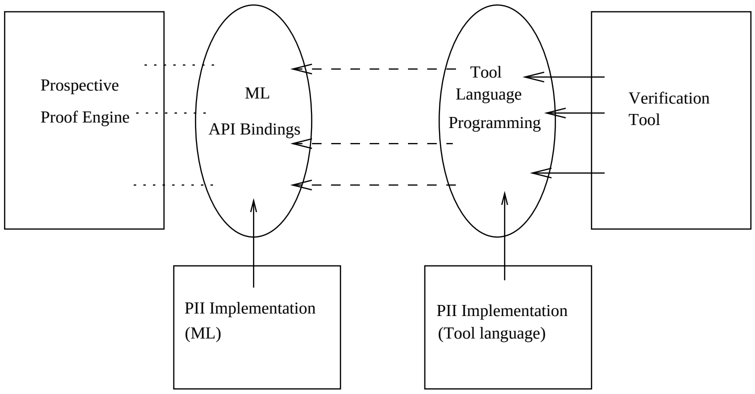

Fig. 5. The Plugin Aspect

It is also possible to run a verification tool in a 'harness' provided by the PROSPER toolkit. This talks to a tool's command line. This allows almost any tool to be used as a plugin, although the tool must be treated as a black box.

4.4 A Complete System

An application developer's view of a complete system should be something like Figure 6. Components are accessible to each other via their APIs. Communication is made possible by low-level functionality irrelevant to the developer. Components can be subdivided into those parts that provide important functionality, databases of functions in the component's API, and modules expressing the functionality of some other component's API.

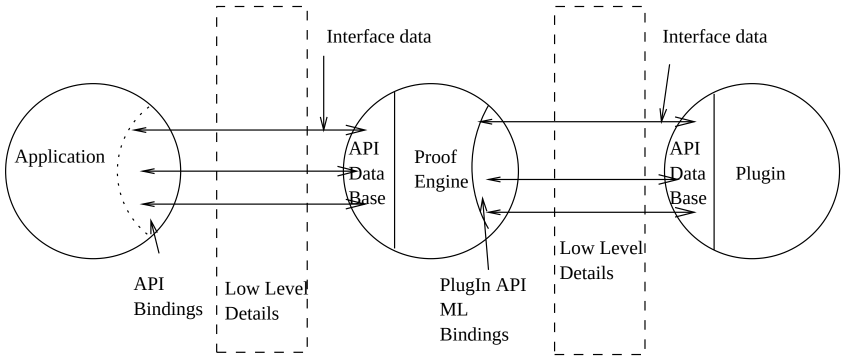

Fig. 6. A complete system

Someone working with such a system can issue an instruction which invokes verification. Such an instruction may arise automatically in response to certain actions they take in the process of design. This instruction states some conjecture which is translated into interface data and passed to a function in the API of the proof engine. This function may, by way of example, break the conjecture down into a number of sub-problems some of which are dealt with by procedures in the proof engine and some of which are passed on as interface data to a function in the API of a plugin. The plugin function executes and returns a result. The proof engine takes the result and may do some more processing on it before passing back its own result to the application. If this is successful and the verification arose automatically the user may not even be aware that anything has taken place. If it is unsuccessful then the user might receive a warning message about the actions they have just performed.

5 Case Study

We present a case study of the use of the Prosper toolkit to embed some simple verification into a well known, existing application.

Excel is a spreadsheet package marketed by Microsoft [18]. Its basic constituents are rows and columns of cells into which either values or formulae may be entered. Formulae refer to other cells, which may contain either values or further formulae. Users of Excel are likely to have no interest in using or guiding mathematical proof, but they do want to know that they have entered formulae correctly. They therefore have an interest in 'sanity checking functions' that they can use to reassure themselves of correctness.

As a simple case study, the Prosper toolkit developers undertook to incorporate a sanity checking function into Excel. We chose to implement an equality checking function which would take two cells containing formulae and attempt to determine whether these formulae were equal for all possible values of the cells to which they refer.

Simplifying assumptions were made for the case study. The most important were that cell values were only natural numbers or booleans and that only a small subset of the functions available in Excel (some simple arithmetical and logical functions) appeared in formulae. Given these assumptions, less than 150 lines of code were needed to produce a prototype. This prototype handled only a small range of formulae, but it demonstrated the basic functionality.

While the resulting program is clearly not marketable (requiring two machines using two different operating systems) it was pleasing to find it so easy to embed some verification into an existing program.

5.1 Architecture

The main difficulty in the case study was that Excel is Windows based, whereas the prototype toolkit had been developed for UNIX machines.[4] A subsidiary

[4] We intend to port a future version to Windows.

difficulty was that the PII was not implemented in Visual Basic, the macro language of Excel.

These problems were solved by using a Python implementation of the PII. Python has library support for COM (a middleware component standard which is also supported by Visual Basic and is a common way to add functionality to Excel). Python also has a freely available socket library, allowing a Python program on Windows to communicate via sockets with a proof engine on UNIX. The decision was taken not to implement the PII in Visual Basic but to call a Python COM server to handle the tasks in the application aspect and communicate both as a client to the proof engine running under Linux and as a server to an Excel COM client running under Windows. The easiest way to access Excel's formulae is as strings. It would have been necessary, whatever the approach taken, to parse these into interface data logical terms and it was unimportant whether this effort was made in Visual Basic or in Python. We hope that a Python based COM component implementing the PII will be of more general interest and use than a Visual Basic implementation of the PII would have been. A view of the architecture is shown in Figure 7.

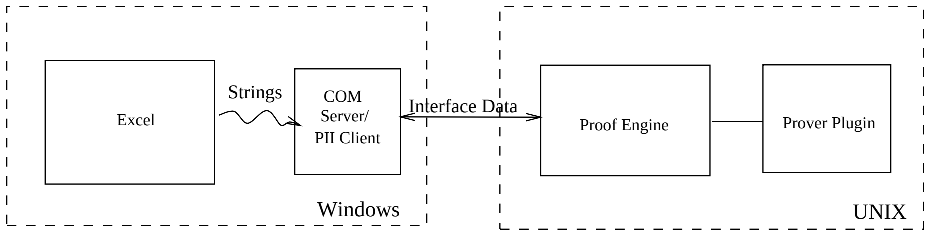

Fig. 7. The architecture of Excel with embedded verification

5.2 The Theorem Prover Aspect

The initial custom procedure is very simple-minded. It uses an arithmetic decision procedure provided by HOL98 and a propositional logic plugin decision procedure (based on Prover Technology's proof tool [23, 22]) to decide the truth of formulae. While the approach is not especially robust, it is strong enough to handle many simple formulae.

This proved to be a very small piece of code (approx. 45 lines of ML were needed to write the function and place it in the API database). A more developed version of such a proof engine would require longer, more specialised code.

5.3 The Application Aspect

A function, `ISEQUAL`, was written using Excel's macro editor. Once written, it automatically appears in Excel's function list as a User Defined Function and can be used in a spreadsheet like any other function. `ISEQUAL` takes two cell references as arguments. It recursively extracts the formulae contained in the cells as strings (support for this already exists in Excel) and passes them on to the Python object.

The Python component parses the strings to interface data logical terms, which it passes on to the decision procedures in the proof engine. It returns the result of the proof attempt as true, false, or 'unable to decide', which is displayed in the cell containing the `ISEQUAL` formula.

The application aspect consisted of roughly 30 lines of Visual Basic code and 30 of Python code. We feel that the case study illustrated the relative speed and simplicity with which a prototype of embedded verification can be produced using the PROSPER toolkit.

6 Related Work

Combined Tools. There are many decision procedures available as verification tools, especially for use with hardware verification. They all have practical limits on the size of design with which they can cope. There has also been a great deal of recent work in combining decision procedures (in particular model checkers) with automated theorem proving to increase the size of design that can be dealt with [1, 15, 20, 21]. The Hardware Verification Workbench, that the PROSPER project plans to produce, will hopefully make use of much of the knowledge and techniques developed for integrating model checkers and theorem proving.

In a slightly different vein the HOL/CLAM project [3] linked HOL to CLAM [5], a proof planning system which specialises in automating inductive proof. The HOL/CLAM project is, in some ways, a predecessor to PROSPER and much has been learned from it.

All this work has focused on producing one customised solution whereas PROSPER hopes to provide a framework in which many such interactions can be investigated.

Integration Architectures. There are several projects that provide a generic framework for the integration of tools.

ΩMEGA [2] is a system developed to act as a mathematical assistant. Like PROSPER, ΩMEGA makes use of other reasoning systems (e.g. resolution theorem provers and computer algebra systems). These are all incorporated into a distributed MathWeb [9] and there is work in progress to produce a standard interface for integrating components.

ETI [24], the Electronic Tool Integration platform, is an ambitious project aimed at allowing both the easy and rapid comparison of tools purporting to do similar jobs, and also the rapid prototyping of combinations of such tools (any software tool, not just verification tools). ETI has its own language, HLL, which acts much like PROSPER's combination of ML and interface data to provide a scripting language for tool integration. It is also possible to automatically generate glue code from easily written specifications. The ETI's implementation is based on C++, which allows all tools written in C++ to be treated in a glass box fashion, just as Prosper allows all tools written in the languages which implement the PII to be treated as glass boxes.

The OMRS project aims to develop an *open* architecture for reasoning systems to be integrated together relatively easily. This architecture consists of three components: the logic of the system [12], the control strategies used by the system [7], and the interaction mechanisms supported by the system. Its framework forces systems to identify clearly what are the sequents, inference rules, control information, etc. and so makes them more open and extensible. The intention is that future reasoning systems will be developed using the OMRS architecture. At the same time work is underway to re-engineer popular existing tools, most notably ACL2 [4], so that they conform to the OMRS specifications.

These systems all allow the integration and combination of verification components ranging from an entirely black box treatment to an entirely glass box treatment in the case of OMRS. We prefer an easier and more flexible approach than OMRS allowing off-the-shelf integration rather than re-engineering. This means it is easier to build an unsound tool with our toolkit. We are not ignoring the logical issues but intend to solve them on an ad hoc basis. ETI is wider in scope but less specific than Prosper. It is forced to treat some components as black boxes, which is inappropriate for many of the interactions Prosper wishes to study. On the other hand, in many cases it is simple to experiment with coordination of several tools using ETI because of its automatic synthesis features.

Design Tools with Embedded Verification. The UniForM project aims to encourage the development of reliable software for industrially relevant tasks by enabling suitable tool-supported combinations of formal methods. The UniForM Workbench [16] is intended to be a generic framework, instantiated with specific tools. The project has produced a workbench for software design that gives access to the Isabelle theorem prover plus other verification tools through their command lines. The various components are held together by Concurrent Haskell, which is used as a sophisticated encapsulation and glue language.

The UniForM project is similar to Prosper, with its focus on the integration of component based verification into design tools, its use of a functional language to manage the various components, and the provision of a theorem prover to perform logical tasks. But, the Workbench is a design tool in its own right rather than a toolkit for embedding verification into a design tool. The Workbench also treats plugin decision procedures as black boxes.

We are not aware of any project, other than Prosper, which easily allows the integration of existing components with the view to producing an embeddable customised proof engine.

7 Conclusions

For embedded (possibly invisible) verification engines to gain widespread acceptance and use, verification tools must be customisable and combinable. We believe the way forward draws on many of the standard aspects of component technology but also requires dedicated support for building custom proof engines, such as language-independent datatypes for communicating logical concepts.

We hope that the work on PROSPER has been a significant step forward in establishing the nature of the support needed to encourage embedded verification. The focus of future work centres around three areas: basic improvements of the underlying implementation; case studies of the effectiveness of the toolkit (we are interested not only in the ease with which theorem proving can be embedded in an application but also in the benefits gained from the combination of theorem proving and decision procedures) and the development of generic proof support for integrated verification (procedures for handling certain classes of plugin effectively, methodologies for ensuring soundness, etc.).

Most importantly, we believe the way to encourage the incorporation of formal verification within design flows is not through the provision of some large *tool* that can perform a wide range of verification tasks but through the provision of a *toolkit* that allows the development of specialised proof engines.

References

1. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. Y. Bertot, G. Dowek, A. Hirshowitz, C. Paulin and L. Théry (eds), *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1690, Springer-Verlag, pp. 323–340, 1999.
2. C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, A. Meirer, E. Melis, W. Schaarschmidt, J. Siekmann, and V. Sorge, ΩMEGA, Towards a mathematical assistant. *14th Conference on Automated Deduction*, W. McCune (ed), Lecture Notes in Artificial Intelligence 1249, Springer-Verlag, pp. 252–255, 1997.
3. R. Boulton, K. Slind, A. Bundy, and M. Gordon, An interface between CLAM and HOL. J. Grundy and M. Newey (eds), *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1479, Springer-Verlag, pp. 87–104, 1998.
4. B. Brock, M. Kaufmann, and J Moore, ACL2 Theorems about Commercial Microprocessors. M. Srivas and A. Camilleri (eds), *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, Springer-Verlag, pp. 275–293, 1996.
5. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill, The Oyster-Clam system. *10th International Conference on Automated Deduction*, M. E. Stickel (ed), Lecture Notes in Artificial Intelligence 449, Springer-Verlag, pp. 647–648, 1990.
6. A. Church, A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.
7. A. Coglio, The Control Component of OMRS: NQTHM as a Case Study. Extended abstract in *Proceedings of the First Workshop on Abstraction, Analogy and Metareasoning*, IRST, Trento, Italy, pp. 65–71, 1996.
8. J. Fitzgerald and P. G. Larsen, *Modelling Systems: Practical Tools and Techniques in Software Development*, Cambridge University Press, 1998.
9. A. Franke, S. M. Hess, C. G. Jung, M. Kohlhase, and V. Sorge, Agent-Oriented Integration of Distributed Mathematical Services. *Journal of Universal Computer Science*, **5**(3), pp. 156–187, 1999.

10. J. A. Goguen and Luqi, Formal methods and social context in software development. *Proceedings TAPSOFT'95*, Lecture Notes in Computer Science 915. Springer-Verlag, pp. 62–81, 1995.
11. M. J. C. Gordon and T. F. Melham (eds), *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.
12. F. Giunchiglia, P. Pecchiari, and C. Talcott, Reasoning Theories: Towards an Architecture for Open Mechanized Reasoning Systems. F. Baader and K. U. Schulz (eds), *Frontiers of Combining Systems—First International Workshop (FroCoS'96)*, Kluwer's Applied Logic Series (APLS), pp. 157–174, 1996.
13. A. Holt and E. Klein, A semantically-derived subset of English for hardware verification. *37th Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference*, Association for Computational Linguistics, pp. 451–456, 1999.
14. J. Hurd, Integrating Gandalf and HOL. Y. Bertot, G. Dowek, A. Hirshowitz, C. Paulin and L. Théry (eds). *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1690, Springer-Verlag, pp. 311–321, 1999.
15. J. Joyce and C.-J. Seger, Linking BDD based symbolic evaluation to interactive theorem proving. *ACM/IEEE Design Automation Conference*, June 1993.
16. B. Kreig-Brükner, J. Peleska, E.-R. Olderog, and A. Baer, The UniForM WorkBench, a Universal Development Environment for Formal Methods. J. M. Wing, J. Woodcock and J. Davies (eds), *FM'99—Formal Methods*, vol. 2, Lecture Notes in Computer Science 1709, pp. 1186–1205, 1999.
17. K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers. 1993.
18. Microsoft Corporation, *Microsoft Excel*, `http://www.microsoft.com/excel`.
19. R. Milner, M. Tofte, R. Harper and D. MacQueen, *The Definition of Standard ML (Revised)*, MIT Press, 1997.
20. J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, First Quarter, 1999. Online at `http://developer.intel.com/technology/itj/`.
21. S. Rajan, N. Shankar, and M. Srivas, An integration of model checking and automated proof checking. *International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 939, Springer-Verlag, pp. 84–97, 1995.
22. M. Sheeran and G. Stålmarck, A tutorial on Stålmarck's proof procedure for propositional logic. *The Second International Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science 1522, Springer-Verlag, pp. 82–99, 1998.
23. G. Stålmarck and M. Säflund, Modelling and Verifying Systems and Software in Propositional Logic. *Proceedings of SAFECOMP '90*, Pergamon Press, pp. 31–36, 1990.
24. B. Steffen, T. Margaria, and V. Braun, The Electronic Tool Integration Platform: concepts and design. *International Journal on Software Tools for Technology Transfer*, **1**(1 + 2), pp. 9–30, 1997.
25. T. Tammet, A resolution theorem prover for intuitionistic logic. *13th International Conference on Automated Deduction*, Lecture Notes in Computer Science 1104, Springer-Verlag, pp. 2–16, 1996.