The PROSPER toolkit

Louise A. Dennis\textsuperscript{1}, Graham Collins\textsuperscript{2}, Michael Norrish\textsuperscript{3,*}, Richard J. Boulton\textsuperscript{2}, Konrad Slind\textsuperscript{3,**}, Thomas F. Melham\textsuperscript{2}

\textsuperscript{1}School of Computer Science and Information Technology, University of Nottingham, NG8 1BB, UK
\textsuperscript{2}Department of Computing Science, University of Glasgow, G12 8QQ, UK
\textsuperscript{3}Computer Laboratory, University of Cambridge, CB3 0FD, UK

Published online: 19 November 2002 – © Springer-Verlag 2002

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 can be treated as components. A system incorporating such tools becomes another component that can be embedded in an application. This paper describes the software toolkit developed by the project. 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.

Keywords: Tool integration – Formal verification – Proof engine – Theorem proving

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 wide range of general properties of their designs. The expectation is that this can give more thorough probing of system quality and more complete (or possibly earlier) detection of errors than testing alone.

However, 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 ambitious academic Formal Methods programme has still to deliver significant benefits to large-scale design practice, and formal verification has, in consequence, been 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 [34].

At present, a developer wishing to incorporate verification capabilities into a CAD or CASE tool, or any application, will face a difficult choice between creating a verification engine from scratch or adapting parts of 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\textsuperscript{1} addressed 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

\textsuperscript{1} http://www.dcs.gla.ac.uk/prosper

\footnotesize
\textsuperscript{1} Work funded by ESPRIT Framework IV Grant LTR 26241.
\textsuperscript{*} Michael Norrish is supported by the Michael and Morven Heller Research Fellowship at St. Catharine’s College, Cambridge.
\textsuperscript{**} Konrad Slind is now at the School of Computing, University of Utah, Salt Lake City UT 84112, USA.
support invisible to the end-user – or at least, more realistically, to incorporate it within the interface and style of interaction they already use.

An early version of the PROSPER toolkit was described in [26]. The present paper describes a more recent release of the toolkit (version 1.4) and the methodology of building systems with it. The paper begins with a description in Sect. 2 of the PROSPER approach, followed by some background information in Sect. 3. Details of the integration mechanism are then given in Sect. 4. Section 5 describes the management of proof-related data and Sect. 6 discusses how the toolkit should be used. More advanced features of the toolkit are presented in Sect. 7. An overview of some case studies is given in Sect. 8, and advice on the applicability of the toolkit is given in Sect. 9. The final sections cover related work and draw some conclusions. Further technical details of the toolkit can be found in the user guide [27].

1.1 Notation used in diagrams

This section provides notes about the notation used in the subsequent diagrams. The reader may wish to skip this and refer back to it later when studying the diagrams. Most diagrams represent either a collection of interacting processes or an aspect of programming using the PROSPER toolkit.

1. Process diagrams
   - Boxes represent executable programs or modules.
   - Boxes which share an edge indicate programs or modules that are tightly coupled (i.e., written in the same language and running in the same process).
   - Basic arrows indicate a client/server relationship between two processes – the direction of the arrow is from the client to the server. These arrows may be labelled with informal descriptions of the interactions.
   - Hollow triangle-headed arrows indicate a communication channel. The direction of the arrows indicates the direction of communication. Optional labels above the arrows indicate the kind of channel (e.g., socket or stdout); optional labels in small font below the arrows are informal descriptions of the message content or datatype.
   - Hollow circle arrows are a standard symbol for a COM interface and are used as such. Again labels in small font below the line are an informal description of the message content or datatype.
   - A line without an arrow along the base of such a diagram is the operating system. The type of operating system is stated below the arrow. Processes appearing above the line are assumed to be running in that operating system.

2. Programming diagrams
   - Boxes with solid lines indicate code modules.
   - Clouds indicate modules that may not exist yet.
   - Arrows indicate that one module may use functions or procedures from the other module.
   - Boxes with dotted lines indicate programming languages and contain modules that are implemented in that language.
   - Circles indicate the work of one programmer or team of programmers; modules within a circle are all programmed by the same person or group of people.

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). This allows components to be operated in a machine-oriented rather than human-oriented fashion.

A proof engine can be built by a system developer using the toolkit provided by the project. A proof engine is based upon a theorem prover (e.g., [36, 37, 46], see Sect. 3) 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 logical term, theorem, and goal (see Sect. 3), 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 [37]. HOL98 is highly modular, which suits the PROSPER approach of building up a proof engine from a set of components (be they HOL libraries or external plugins). It also contains a number of sophisticated automatic proof procedures. HOL’s command language is ML [54] (a strict functional programming language) extended with the language of higher order logic [21] and functions to support theorem proving. 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. In addition, there is a database component which duplicates the logical theories stored in the proof engine so that plugins (and possibly applications) can access theory-related data (see Sect. 3) while the proof engine is busy. The proof engine can both upload data to the database and read from it; plugins and applications can only read data. More details about this can be found in Sect. 5.

The application, proof engine, database, and plugins act as separate components in the final system (Fig. 1). In the current prototype they are also separate processes. Communication between them is treated in a uniform manner specified by the PROSPER Integration Interface.

This technology has been used to add verification capabilities to IFAD’s VDM-SL Toolbox [29]. The PROSPER project also built a Hardware Verification Workbench. This allows specifications in Verilog and VHDL to be checked by a proof engine that incorporates a model checker.

Example 1. The IFAD VDM-SL Toolbox [29] is a software design tool supporting the specification language, VDM-SL. Its functions include syntax and type checking of specifications, and code generation in C++ and Java. The 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 try to check their truth.

Many invariants, however, 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 the following components: 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 integrated together into the IFAD Toolbox using the PROSPER toolkit.

3 Background

This section provides some background information about theorem proving software, specifically the HOL system [37]. Readers familiar with HOL or a similar system may skip this material.

A theorem prover is a computer program that assists in proving theorems in some logical formalism such as predicate calculus (first-order logic). Some theorem provers are fully automatic while others require some degree of user interaction. All theorem provers perform logical deduction by applying inference rules of the logic, though the extent to which the inference rules are made explicit varies between systems. This is in contrast to model checkers, another class of verification tool, which exhaustively explore state spaces. A typical inference rule is modus ponens which deduces the logical formula \( B \) from the two formulae \( A \implies B \) and \( A \).

The HOL system, which is used in the PROSPER toolkit, has a fairly expressive higher order logic [21] in which quantification can occur over functions and functions can take other functions as arguments. A typical higher-order theorem is

\[
\vdash \forall f. \forall g. (\forall x. f x = g x) \supset (f = g)
\]

Fig. 1. A system built with the PROSPER toolkit
which states that if two functions \( f \) and \( g \) return the same result for all possible arguments, then they are the same function. The turnstile symbol \( \vdash \) denotes theoremhood.

HOL is implemented in ML, a strongly typed functional programming language. In the HOL system, new ML types are defined for logical terms (expressions), the logical types of terms (e.g., the term “1 + 2” might have the type “num”, for “number”), and theorems. HOL follows the approach pioneered in LCF [36] where an abstract ML type for theorems (thm) is used to ensure that a theorem can only be generated by sound logical steps.

The abstract type for theorems consists of some basic theorems for the axioms of higher order logic together with ML functions for the primitive inference rules. A new value of type thm can only be generated from existing theorems using the inference rules. For example, the ML function for the modus ponens rule takes two existing theorems and returns a new theorem provided the original theorems have the right form. If they do not, an ML exception is raised. By this means a programmer is prevented from implementing new inference rules that do not ultimately use the primitive inference rules.

Derived inference rules can be written to perform larger logical steps. Eventually, the steps performed become so large that it is more appropriate to call the new inference rules proof procedures. HOL has a large library of inference rules and proof procedures built-in, including decision procedures, which are proof procedures that are guaranteed to deduce that a given formula (a Boolean-valued term in HOL) is either true or false provided the formula is in some well-defined subclass of the logic. More generally, proof procedures may fail (raise an ML exception) because full higher order logic is undecidable.

Due to the undecidability of higher order logic and the consequent inability to have a fully automatic proof procedure for it, HOL provides facilities for interactive (viz. user-guided) proof. The most widely used facility is the subgoal package which implements a backwards style of proof. Rather than building up the desired theorem from simpler theorems using inference rules, backwards proof starts from the formula to be proved (called the goal) and reduces it to simpler subgoals using ML functions called tactics until the subgoals can be proved automatically.

A tactic takes a goal as its argument and returns a list of subgoals together with an inference rule that can prove a theorem for the original goal if given theorems for the subgoals. The subgoal package orchestrates the presentation of the goals to the user and when all subgoals have been proved it applies the inference rule to yield the theorem for the original goal.

A common form of inference rule, used for equational reasoning, is rewriting. Rewriting uses a theorem of the form \( L = R \) to soundly change a subterm of a term. For example, the term “\( x \times ((y \times z) + 0) \)” can be rewritten to “\( x \times (y \times z) \)” by applying the theorem “\( A + 0 = A \)” (a rewrite rule) to the subterm “\( (y \times z) + 0 \)”. The left-hand side of the rewrite rule is matched against the sub-

term so that the variable \( A \) becomes instantiated to “\( y \times z \)”. This instantiation is used to replace all the variables in the right-hand side of the rule to produce the new subterm.

Of course, for the HOL system to be of use in any serious proof development, it must be possible to store newly-proved theorems for later use. HOL provides a theory mechanism for this. A HOL theory consists of, amongst other things, the definitions of new constants and new types together with theorems proved about them. Theories are organised in a directed acyclic graph according to which other theories they depend on. A theory together with the transitive closure of its predecessors in the graph is known as a theory hierarchy.

In keeping with PROSPER’s focus on a component-based architecture, the latest version of the HOL system, HOL08, has been designed to be highly modular. Most of the proof procedures and theories, even commonly used ones, are not included in the default executable but instead are loaded on demand.

4 The PROSPER Integration Interface

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

The PII is not a specification language, nor is it an interface definition language (IDL) like the IDLs of middleware standards (e.g., CORBA [56]). The PII is, however, similar to the entities that IDLs are used to describe; the PII gives the datatypes and operations available to a programmer using the PROSPER toolkit. In other words, the PII is like an application programming interface (API), but we prefer to avoid using that expression because it can cause confusion with the more specific APIs that are built using implementations of the PII.

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. The interface data datatype is essentially a disjoint union of several simpler datatypes — some of which denote standard sets of values, such as strings or numbers, and some of which denote special sets of structured “logical data”, such as formulae in the formal language of logic.

The formal logic components of interface data comprise a complete representation of the formal language of higher order logic used by HOL, so any formula expressible in higher order logic can be passed between components. Many plugins operate with logical data that is already a subset of higher order logic (e.g., first-order or propositional logic) or embeddable in it (e.g.,


4.1 Interface data

Viewed abstractly, interface data is a language for communicating data values between components of systems built using the PROSPER toolkit. The PII gives an abstract specification for the operations that construct and manipulate interface data, but the exact form of these operations and their usage depends on the programming language in which the PII is implemented. The data itself can be described using a grammar. A grammar for the concrete form used to pass the data between components is given in the PII implementor’s guide [15], but a normal user of the PROSPER toolkit should never have to see this.

Interface data is a disjoint union of several other types. These include standard types such as Booleans, integers, and strings as well as lists (allowing tree structures to be communicated). These types are used for communicating general data between components. For communicating logical information, e.g., formulae of higher order logic, interface data also includes special types for logical terms, the types of logical terms, and theorems.

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

Example 2. The PII specification has the following operations for strings:

\[
\begin{align*}
\text{mk_string} & : \text{string} \rightarrow \text{interface_data} \\
\text{dest_string} & : \text{interface_data} \rightarrow \text{string} \\
\text{is_string} & : \text{interface_data} \rightarrow \text{bool}
\end{align*}
\]

where “:” means “is of type” and “\(\rightarrow\)” denotes a function type. (We also use “\(*\)” for Cartesian product.) Assuming the implementation language already contains types for strings and Booleans, all that is needed is a type for interface data and then an implementation of the three operations such that mk_string takes a string as input and returns interface data, dest_string takes some interface data as input and returns a string and is_string takes some interface data as input and returns true or false. Of course these operations are expected to behave in a sane fashion, for example:

\[
\text{dest_string} (\text{mk_string} s) = s
\]

The Java implementation has a class for PII strings which implements a Java interface for PROSPER interface data. The class has a public constructor PII_String together with several methods, some of which are specific to the string class and others of which are generic to the interface. The methods include one for obtaining the underlying Java string from a PII string and one for testing whether a PII object is a string.

The ML implementation has a recursive datatype for interface data with constructors for each kind of data. The programmer can use either pattern matching on the constructors or the functions mk_string, dest_string, and is_string that are also supplied. The function dest_string raises an ML exception if applied to the wrong kind of interface data.

The C implementation uses a structure to represent interface data with a union of the types needed to represent the different kinds of interface data and an enumerated type to tag which kind a particular instance represents. There is a function PII_mk_string which
allocates memory and stores the string (char *) argument. The PII_dest_string function simply extracts
the string value and PII_is_string tests the tag to see
if its value indicates that the PII data is a string. As C
is not a garbage-collected language, operations are also
included to free the memory allocated for PII structures.

The programmer uses strings in the normal way for
each language. Differences between languages, such as escape
sequences, are handled by the translation layer of the
PII, which ensures a consistent encoding for transport
data between components. Only 8-bit characters are
supported.

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 and are the basic
expressions for communicating logical information (e.g., conjectures that need proving). As usual with Church-
style formulation [21], there are four syntactic categories:
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. The basic
constructors and type relation operations for terms are
shown in Fig. 3.

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 con-
ists of the usual logical connectives and quantifiers plus
a few other common constructs.

A programmer using the PII usually needs to have
knowledge of the abstract syntax of higher order logic
terms, but knowledge of the particular concrete syntax
used by HOL98 will be necessary only if the programmer
plans to use the proof engine interactively, e.g., for test-
ing. While working on the application side or plugin side
of the interface, the concrete syntax is definitely not re-
quired. Furthermore, in many situations, the actual terms
used will appear just like formulae of the more widely
known first-order and propositional logics.

\begin{verbatim}
mk_vartype : type_name -> term_type
mk_type   : type_name * term_type list ->
            term_type

type_of   : term -> term_type

mk_var    : var_name * term_type -> term
mk_const  : const_name * term_type -> term
mk_comb   : term * term -> term
mk_abs    : bound_var * term -> term
\end{verbatim}

Fig. 3. The basic constructors and type relation operations for
terms

4.2 API Support

The PII application support layer provides functions to
allow client and server components to handle remote pro-
cedure calls. It uses a datatype, interface_data_result,
with constructors

\begin{verbatim}
mk_succeeded : interface_data ->
               interface_data_result

mk_failed   : interface_data ->
               interface_data_result
\end{verbatim}
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 argu-
ments. It returns an interface data result.

A server has some database manipulation functions
for using an API database containing functions of type

\begin{verbatim}
interface_data -> interface_data_result
\end{verbatim}

referred by strings. These are used to process calls from
a client. This is the basic operation for the remote ma-
nipulation of components in a system built using the
PROSPER toolkit.

There are three special interface data constructors for
handling functions:

\begin{verbatim}
mk_ref : string -> interface_data

mk_app : interface_data * interface_data ->
        interface_data

mk_fun : (interface_data ->
         interface_data_result) ->
         interface_data
\end{verbatim}

PROSPER does not support the passing of algorithms or
code fragments between components so it is impossible
for function values to be passed between components in
the same way as values of other types. However, it seems
overly restrictive to disallow a client program from in-
structing its server to combine functions together. For
instance, a client might want a server to perform a se-
quence of commands, the output of one acting as the
input of the next. If the output were a large data structure
then considerable time could be wasted passing it back to
the client only to have it sent again to the server. For this
reason, servers allow functions to be referenced by strings.
In the simplest case the expression

\begin{verbatim}
mk_app (mk_ref (fun_name), idata)
\end{verbatim}

(where fun_name is a string and idata is a list of inter-
face data) applies the function referenced by the string
fun_name to the list of arguments, idata. All functions
that can be accessed in this way are available in the
database.

It is possible for a function in the database to return
another function as its result. These are constructed using
the mk_fun constructor which is only available to a server
and cannot be used by the client. In this way, if a function
is returned by a database query it may be used by other
procedures in the server, but may not be passed back to the client.

The most important use of these functions is in exploiting tacticals which take tactic functions as arguments. Tactics are used to decompose a conjecture (a goal) into sub-problems. Tacticals combine existing tactics to form new tactics. For example, the tactical

\[ \text{THEN : tactic } \times \text{ tactic } \rightarrow \text{ tactic} \]

takes two tactics \( t_1 \) and \( t_2 \) as its arguments and returns a new tactic. The new tactic has the effect of applying \( t_1 \) to a goal to produce some subgoals and then applying \( t_2 \) to each of the subgoals to produce new subgoals, i.e., it sequences the tactic operations. Tacticals are a powerful technique for building up customised automated theorem proving tools in HOL which the project did not want to sacrifice.

4.3 Connection support and lower layers

The PII application support layer includes client-side functions for connecting to and disconnecting from servers. The low-level details of communication handling are only relevant to those wishing to implement the PII in a new language.

The underlying communication is currently based on Internet sockets. The decision to use sockets and not a middleware standard such as CORBA [56] was based on the lack of CORBA (or other middleware) support for some of the languages we planned to use. It is possible that at a future date the PROSPER toolkit could switch to the use of a middleware standard. If so, we anticipate that the changes would occur below the application support layer with little change being visible to users of the PII.

5 The PROSPER database

Central to every PROSPER-based system is an ML process that includes at least some minimal theorem proving functionality from HOL98, as well as links to plugins. This central theorem proving process will maintain, as part of its internal state, information about the logical theory context in which the proof engine is operating. This includes the names and types of all logical constants in scope, their formal definitions, and any proved theorems that have been saved for later use.

Parts of this centrally-maintained theory information may also be needed by plugin proof tools. For example, a plugin that does some rewriting (e.g., the ACL2 plugin [68]) may need access to constant definitions in the central proof engine component. Plugins may also return logical data for updating the theorem prover’s theory state, or they may return other values that are interpreted as “requests” for theory information to be passed to them.

Most theory information is just logical data and can, of course, be passed as PII interface data in a plugin function-call or returned as a result. The PROSPER toolkit, however, also provides a more flexible mechanism for sharing theory data via the PROSPER database. This is a separate server process that runs independently of the proof engine and can be communicated with via the standard PROSPER communication protocol. It can be viewed as an externalisation of a great deal of the internal theory state of the proof engine, including the complete hierarchy of logical theories underpinning the application.

The database accepts queries for theory information from any PROSPER client. It also accepts both queries and update requests from the proof engine (see Fig. 1). In actual use, the database is dynamically updated as the theory state of the proof engine changes, with the effect that the database closely mirrors this state at all times. Clients such as plugins can then access this information independently of the central proof engine process.

In practice, the database plays two important roles. First, it provides a separate avenue for clients to access valuable information in the proof engine theory state, and thus relieves the central proof engine process of the computational burden of servicing requests, allowing it to focus better on proof activities. Second, the database allows plugins to asynchronously access the state of the theorem prover without fear of deadlock. (ML may block waiting for a plugin to return an answer.)

The database responds to a simple but reasonably powerful range of queries. For example, one can use higher-order term pattern matching to search the entire theory hierarchy for matching axioms, definitions, and theorems. In principle, the queries supported by the database enable the entire theory hierarchy of HOL to be copied to a remote location.

6 Using the toolkit

The basic PROSPER toolkit consists of relatively little: an extension of the core of HOL expressing some of its functionality in an API database and an ML implementation of the PII. This allows HOL to be used as a very simple Core Proof Engine. It also allows a developer to write extensions to HOL (see Sect. 3) and place them in a custom API.

The Core Proof Engine contains an ML implementation of the PII (with the application support layer visible); the HOL logical inference rules; a type \texttt{thm} for theorems and a mechanism for “tagging” where a theorem has come from. This is the minimum needed for a program to be considered a theorem prover. Readers used to LCF-style theorem provers ([36] and Sect. 3) should note that there is no provision of tactics in this minimal core, although they can be added as extensions.

The modular nature of HOL allows a toolkit developer to simply load in the Core Proof Engine as a module.
in their own customised proof engine code to give them a minimal starting point. Similarly, other support modules supplied by the toolkit (e.g., the proof management module, see Example 4) can be easily loaded into new proof engines. Scripts are provided to allow proof engines to be compiled easily and run in a uniform manner.

Many applications will require a version of the PII in an implementation language other than ML. The toolkit currently includes PII implementations in Java and C, and code to construct some plugins (a version of the SMV model checker [51], Prover Technology’s Prover Plug-In [65, 67], and AC/3 [42]) which can be added into proof engines. Third party plugins are also available for ACL2 [46, 68], Gandalf [44, 73] and SVC [70]. As described in Sect. 5, there is also a database component that shadows the proof engine, allowing plugins to access theory-related data while the proof engine is busy.

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: the theorem prover aspect, the application aspect, and the plugin aspect. These partition the effort into the areas most likely to be undertaken by different people. The three aspects also help to identify which parts of a final system should be responsible for which tasks.

6.1 The theorem prover aspect

The theorem prover aspect (Fig. 4) is concerned with producing a custom proof engine. Typically this will include some application-specific logical definitions, theorems, and proof procedures. These procedures may well use plugin decision procedures (e.g., for predicate or propositional logic) or even include, as plugins, verification tools previously developed 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. A full programming language, the functional programming language ML [54], is available for this development. The resulting proof procedures are made available to the intended application via an API. This API can be passed on to the developer of the application.

The strong typing of ML, together with HOL’s way of representing theorems (see Sect. 3), ensures that only sound logical steps can be implemented. The soundness is, however, relative to the consistency of any plugins whose results are being trusted, i.e., plugins that are being used as “oracles”.2

The toolkit comes with ready-made modules that contain some of the functionality included in HOL’s libraries (of theories and proof procedures) and the interfaces to some plugins. Any logical theories used in the proof engine must be uploaded to the database so that they can be accessed by other components while the proof engine is busy.

All implementations of the PII provide a basic mechanism by which a client can invoke any registered function in a proof engine’s API. (This is the call_server function and associated “API database” of Sect. 4.2.) However, any serious application will want to wrap up the API with functions directly in the client application’s own programming language that provide a more direct expression of the operations available.

We call the implementation of these operations language-specific bindings. The purpose of language-specific bindings is to hide all instances of the call_server primitive from the application aspect developer, and to give

---

2 HOL provides a “back door” for creating theorems without using the inference rules. This allows results from plugins to be imported. The theorems (and any other theorems derived from them) are tagged as having been created in this way.

---

Fig. 4. The theorem prover aspect
a clean interface to the embedded proof capability in the application’s own terms. This can, of course, be done only when the implementation language of the target application is known.

The exact form the language-specific bindings for an application will take depends on the programming language. For example, in an object-oriented language it would be natural to present the operations available in a proof engine’s API as methods in some verification class. In a functional language, the operations will be presented as a collection of functions.

Example 3. 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 Sect. 10). These combined procedures will be developed in the theorem prover aspect and presented as the API to a custom proof engine.

Example 4. One of the modules that comes with the toolkit is a proof management module. This allows proofs in a proof engine to be manipulated (e.g., by switching between open goals and partial proofs, backtracking, restarting a proof, etc.). If this module is incorporated in a proof engine then that proof engine will offer such functions in its API. The API, however, tends to force the programmer to use a functional style, which is not appropriate in, for instance, an object-oriented language such as Java. In Java it would be more natural to have a class of objects for proofs with the individual operations that call the proof engine as methods in this class. To achieve this, some programming is required on the Java side of the interface which does not necessarily have anything to do with the application. The results of this programming effort are the “language-specific bindings” for this API (see Fig. 5).

6.2 The application aspect

The application aspect (Fig. 6) 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.

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

Example 5. The project is investigating the use of a natural language interface [43] to its Hardware Verification Workbench that translates statements about circuits, in the normal technical language of engineers, into CTL propositions [51] that a proof engine can verify. A given natural language specification may be ambiguous and so give rise to more than one CTL formula. To disambiguate, timing diagrams (waveforms) are generated from the CTL formulae and presented to the engineer to select which interpretation she or he intended by the natural language specification. In this way, engineers never have to see CTL formulae; they only work with things with which they are already familiar, namely natural language and timing diagrams.

It is anticipated that most applications will require a combination of both the application and theorem prover aspects. Although we have here suggested that all the verification functionality in the theorem prover aspect take place in ML and be focused on providing a custom proof engine, it is not impossible to migrate some theorem proving tasks into the application if so desired. Similarly it would also be possible to place in the proof engine some of the burden of translating the output into the application’s concepts. However, we feel that the separation of the tasks described here is the cleanest.

6.3 The plugin aspect

The third aspect of using the PROSPER toolkit is development of plugins (Fig. 7).

A plugin developer programs both in ML and in the plugin’s own implementation language. The developer’s

---

3 For instance, interface data includes strings, so messages to the user can be constructed in the proof engine.
job is to make selected entrypoints into the plugin’s functionality available to PROSPER clients via the PII machinery for building APIs (Sect. 4.2). These entrypoints will typically be functions or procedures within the plugin that offer useful, coherent reasoning capabilities to clients. The top-level function for invoking a decision procedure on a given problem would be a typical example. A more fine-grained function might be one that just sets some flags or operating parameters (plugins can have state).

Of course, the underlying software for a plugin will have its own internal data structures for logical expressions and so on. A plugin developer must therefore create, in the plugin’s implementation language, any translations required to map between PII interface data and the data structures for the arguments needed by these functions.

There will also be some work to do in ML on the HOL theorem prover side, from which the plugin will be invoked. In particular, the plugin’s API will have to be expressed as functions via some ML language-specific bindings. In addition, any logical theories required to support the formalism of the plugin will have to be provided by the plugin developer. Typically, this will mean making an embedding of the logic used by the plugin into higher order logic. This can be relatively trivial (e.g., embedding propositional logic) or very difficult, depending on the mathematical world of the plugin.

As can be seen from the above, the plugin aspect is analogous to the theorem prover aspect – except that the required functionality for the API is already implemented (by the tool being made into a plugin) and provision of language-specific bindings is strongly recommended, since the target language is known to be ML.

A plugin can access data about the embedded logic and current theory context from the PROSPER database even when the central proof engine process is waiting for the plugin to respond. This relies on the proof engine having uploaded the theory-related data to the database, and of course the plugin developer must also implement this functionality into the verification tool’s code.

Finally, if the source code for a verification tool is not available, it will not be possible to make it into a plugin as just described. In this case the verification tool may run in a “harness” that talks to the tool’s command line (e.g., through standard input and output in Unix). Such a harness is provided as part of the PROSPER toolkit, and several plugins have been developed with it [44, 68]. However, if source code for the tool is available, it is far better to use the PII for integration. Using the PII avoids the need for expensive and potentially unreliable parsing of the human-oriented output from the tool. It also allows the tool to be treated as a glass box, though how significant this is depends on how much functionality the tool makes available in its command-line interface.

**Example 6.** SVC [72] is a decision procedure for, among other things, linear arithmetic on the rationals. SVC is written in C++. The PROSPER project constructed a plugin from SVC by writing additional C++ functions to translate interface data into SVC’s internal formula syntax and to place an SVC function `check_valid` into the plugin’s API. `check_valid` takes a formula as input and returns true or false. In HOL, ML language bindings are used to wrap up `check_valid` so that its result is presented as an equality theorem stating whether or not the original formula is true for all inputs. A more detailed description of this plugin can be found in Sect. 8.2.

### 6.4 A complete system

An application developer’s view of a complete system should be something like Fig. 8. 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 irrelevant to the developer. Components can be subdivided into those parts that provide important functionality irrelevant to the developer. Components can be subdivided into those parts that provide important functionality irrelevant to the developer.

The database component is not shown in the diagram. From the application developer’s point-of-view it can normally be thought of as part of the proof engine. In certain circumstances, however, an application might need to access theory-related data, in which case the database should be made explicit in the 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.

7 Interrupts and monitoring

In practice it is not sufficient for a proof engine to accept a request from an application and then return a result only when it is ready. With the often unpredictable running time of many proof procedures, it is important that the proof engine (and likewise, any plugins) can be stopped by the end user. Setting a time limit on a computation is a partial solution but it is not really satisfactory. For example, the user may have started the computation unintentionally (say, by clicking on the wrong button of a graphical user interface) and wish to stop the computation immediately.

If all components of the system are running on the same computer, the interrupt mechanism of the underlying operating system can be used (at least for operating systems like Unix). We did not, however, want to impose this restriction.

Another issue is what to do with output from a component that is running on a remote computer. The output from components may be important, for example, in giving the user a sense of how a lengthy computation is progressing. Thus, simply discarding the output is not a satisfactory option.

7.1 The two-process architecture

Instead of using operating system interrupts, a PROSPER application can issue an interrupt instruction using the PII. This is turned into an interrupt signal on the remote computer on which the component is running. To achieve this PROSPER uses an additional process that acts like a proxy for a component. This monitor process can receive the interrupt instruction while the component process is busy. The monitor, which runs on the same computer as the component, uses the operating system to interrupt the component. Typically, a component cannot listen for the interrupt instruction itself because most theorem provers, model checkers, and other verification tools, are single-threaded. If it is busy performing a proof procedure it cannot also be listening for an interrupt instruction.

In addition to handling interrupts, the monitor process captures the standard output streams of the component and redirects them elsewhere. It is the monitor process that communicates with the outside world and all interactions with the component go through it. The connection between a component process and its monitor is illustrated in Fig. 9.

The component is executed by the monitor process so that the monitor can establish all the connections it needs. To the outside world the monitor presents a socket using the PII protocol. Thus, to a client it appears as if the monitor is the component itself. The monitor may also present another socket (for output only) on which output from the component is redirected.

Little has to change in the component itself. It communicates over a socket as if the client was connected to it directly. The only difference is that it will never receive an interrupt instruction. The monitor process forwards most requests and results without taking any special action. Only when it receives an interrupt instruction does it do anything special. Instead of forwarding it to the component, it issues an interrupt signal using the operating system. The component must,
of course, handle the interrupt signal in an appropriate way. Lines of output from the standard output and error streams of the component are tagged before being multiplexed on the logging socket. This merged output can be viewed directly at the application or, subject to suitable software at the application end, it could be split up again into separate streams. Alternatively, the output could be sent to some standard recipient that displays all output to the user.

There is an issue of propagation of interrupts to plugins. It is not appropriate to leave a plugin running when the proof engine has been interrupted. This issue is dealt with by requiring the interrupt handler in the proof engine to send interrupts to all active plugins, but other strategies are possible.

### 7.2 The communication manager and name server

In the above description each PROSPER server component has its own monitor process. In fact, the current architecture has a single monitor process which handles the monitoring for all PROSPER components on its machine. We call this monitor process the communication manager [27, Chapter 8]. A communication manager on one computer should be able to co-ordinate with communication managers on other computers, but at the time of writing that feature is not implemented. It is, nevertheless, possible for a client running on one machine to access (and interrupt) a proof engine and plugins running on another machine.

A possible advantage of having a single process doing all the monitoring is that it can keep track of all the interactions between components and hence automatically propagate interrupts to plugins, but we do not currently take this approach to interrupt propagation (see above).

The communication manager includes a name server facility whereby components may be requested by name instead of by location. This is achieved by means of a database of component names together with information about the components’ initialisation scripts and configuration.

### 7.3 Related work

A very similar solution has been developed by Bertot and colleagues for their graphical user interface to the Coq theorem prover [11, Sect. 3.2.4]. Like the PROSPER team, they found that the single-threaded nature of most theorem provers was a problem. They have a client/server model using socket-based communication with an interrupt facility. Two bidirectional sockets are used. One carries standard input and standard output. The other carries standard error and a message channel from the client to the server (e.g., for interrupts).

This is a bit different from the PROSPER architecture, where standard input is not used; instead there is direct communication over the socket with an API. The PROSPER two-process architecture uses only one socket plus an error log. Another difference that might be significant is that PROSPER needs to be able to deal with black-box components. In the CtCoq work the researchers had full access to the internals of Coq.

In [11], Bertot goes on to discuss the possibility of getting into a confused state when Coq is interrupted while sending data. He suggests a solution based on protecting regions of code from the interrupt so that the interrupt is noted but not acted upon until the output routine has reached a consistent state.

### 8 Case studies

We present three case studies of the use of the PROSPER toolkit in order to illustrate the various aspects of the project. The first study concerns the addition of a simple proof operation to a well-known application, Microsoft Excel. This illustrates the theorem prover and application aspects. The second describes how the Stanford Validity Checker, an automatic verification tool, has been made into a PROSPER plugin. The third covers the use of a plugin in hardware verification.

#### 8.1 Embedding verification into Microsoft Excel

Excel is a spreadsheet package marketed by Microsoft [53]. 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 [24]. 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 that the basic functionality was achievable very easily.

While the resulting program is probably 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.
8.1.1 Architecture

The main difficulty in the case study was that Excel is Windows based, whereas the prototype toolkit had been developed for Unix machines. A subsidiary 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). 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. A view of the architecture is shown in Fig. 10.

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.

8.1.2 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 Prover Plug-In [65, 67]) 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; about 45 lines of ML were needed to write the function and place it in the API database.

8.1.3 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 into interface data logical terms, which it passes on to the decision procedures in the proof engine. It relays back from the proof engine 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.

8.2 The SVC plugin

The Stanford Validity Checker (SVC) has been in development at Stanford University for several years, and is used both in research and as a formal hardware verification tool [7, 49]. SVC allows users to check formulae by means of decision procedures for a subset of first-order logic. The subset includes Booleans, uninterpreted functions and linear arithmetic. There are also some other interpreted functions such as operations on arrays and bit vectors. In this section we summarise how SVC was made into a PROSPER plugin [70].

SVC is written in C++. An implementation of the PII already existed in C and it was a relatively trivial exercise to adapt this to C++. In fact, much of the relevant code was simply imported directly into C++ without alteration. All the work required was standard reworking of C code to allow it to be called from C++.

SVC has a number of access points through its command line interface. Its primary command for proof is check_valid. The user supplies an expression to this function and it returns a result based on the validity of that expression. The result can be VALID, INVALID or INVALID with a counter-example.

8.2.1 Translation between interface data and SVC’s formula syntax

SVC has a large formula syntax for arithmetic, records and bit-vectors. We chose to cater for only the subset of this language covering arithmetical expressions.

The translation from interface data into SVC formula syntax depends upon the representation of rational numbers in interface data terms. To determine this we needed
to know how the rationals were to be represented in HOL.
There is, in fact, no suitable theory for rationals in HOL
and so we had to provide our own (see Sect. 8.2.2).

The translation on the C++ side of the interface is
performed simply, by taking a PII term, and identifying
the kind of term we have (equality, variable, application,
etc.) using the PII’s query operations. With this infor-
mation, the appropriate SVC expression is then built up,
using constructors provided by SVC.

8.2.2 The theory of rationals

SVC assumes that all numbers appearing in arithmetical
expressions are rational numbers. There is no full theory
of rationals in HOL although Harrison provided a theory
of half-rationals to support his construction of a theory of
real numbers [41]. It is preferable to provide definitional
theories for HOL (i.e., constructing a new theory from an
old one by defining the types and constants of the new
theory and then proving that the axioms follow from the
definitions). However it is possible, and in many cases
simpler, to provide an axiomatic theory, which we chose
to do in this case. Our axioms for rationals are based upon
the description in [71].

The theory provides a new HOL type, \( \text{rat} \), with which
to represent rational numbers. It provides the usual con-
stants representing addition, multiplication and inequal-
ities, and asserts standard axioms for these. With this
theory in place we were able to handle a reasonable subset
of SVC’s input language, although currently bit-vectors
and records are not supported.

8.2.3 ML language bindings for the SVC API

On the ML side we implemented two HOL entry points
based upon \( \text{check_valid} \). The first is of type \( \text{term} \rightarrow \text{term} \). The user supplies a term that represents a Boolean
statement (i.e., a formula). This statement can be com-
posed of Boolean and rational expressions. The term is
sent to the plugin using the PII where validity checks are
performed. Assuming an error does not occur, another
term is returned back across the PII. This new term is one
of true, false, or a term representing the counter-example
calculated by SVC. (This counter-example is presented in
the same syntax as the initial term provided by the user.)

The \( \text{check_valid} \) function provides an SVC-centric
entrypoint which is not much use for proof in HOL.
A more HOL-centric entrypoint is \( \text{svcprove} \) which
returns a HOL theorem that can be saved and recalled
when the result is needed at a later date. This function is essentially a wrapper for \( \text{check_valid} \) which con-
structs a HOL theorem based on whether the result is
true or false. Among other things, this theorem can be
used for rewriting sub-expressions. If the result from SVC
is a counter-example, then the original term is equated
to false, and the counter-example is provided as an as-
sumption to the theorem. For example, if trying to prove
\( x = y \) for all rational values of \( x \) and \( y \), a counter-
example might be \( x = 0, y = 1 \). The theorem representing
the counter-example would have assumptions \( x = 0 \) and
\( y = 1 \), and the conclusion \( x = y \) = false.

8.3 Combining verification and circuit debugging in the
hardware verification workbench

The Hardware Verification Workbench is one of the main
case studies being performed by the PROSPER project.
Here we look at a part of that study in which verification
is combined with a plugin for error detection.

The AC/3 circuit rectifier [42] is a tool for performing
automatic error correction in combinational circuits. Two
circuits must be provided to the system, where one serves
as the specification circuit and the other as the current
implementation. AC/3 tries to prove that the two designs
are equivalent and performs automatic error correction if
they are not.

AC/3 has been wrapped up as a PROSPER plugin. Like
many plugins, its ML bindings provide a function that
takes a HOL term of the form \( X = Y \) where \( X \) and \( Y \) are
propositional formulae, and returns the term as a theo-
rem if the two expressions are logically equivalent. If \( X \)
and \( Y \) cannot be proven equivalent, the rectification
engine is invoked and a theorem \( \neg X = Y' \) is returned
where \( Y' \) is logically equivalent to \( X \) and syntactically similar to
\( Y \). In other words AC/3 tries to modify \( Y \) with a mini-
mal number of changes such that both formulae become
equivalent.

9 When and how to use the PROSPER toolkit

In this section, we discuss some practicalities of adopting
the PROSPER toolkit as a means of adding proof support
to an application. However, we wish to emphasise that
PROSPER was a research project and the toolkit is not the
last word on integrating proof support, nor is it developed
to commercial software standards. The following points
should also be remembered while reading this section:

- The PROSPER toolkit is just that, a toolkit; it is not
  an out-of-the-box solution to adding proof support to
  an application. It is intended to make that task easier,
  but in some cases an ad hoc solution may be quicker
  and give as good results.
- We expect different aspects of an integration to be
  performed by different groups of people, e.g., we
  would normally expect a model checker to be given
  a PROSPER interface by the developers of the model
  checker not by the developers of an application who
  want to integrate the model checker into their ap-
  plication. The usability of PROSPER ultimately de-
  pends on formal verification tool developers providing
  a PROSPER interface, so that application developers
  can use such tools “off the shelf”.

\( \frac{x}{y} \) (for all rational values of \( x \) and \( y \), a counter-
example might be \( x = 0, y = 1 \). The theorem representing
the counter-example would have assumptions \( x = 0 \) and
\( y = 1 \), and the conclusion \( x = y \) = false.)
The advice in this section refers to version 1.4 of the toolkit, implementations of which exist for the Linux and (Sun) Solaris operating systems. A description of what is provided in the toolkit can be found in Sect. 6. There is also a user guide [27]. The discussion below focuses on the perspective of an application developer. There are other perspectives on using the PROSPER toolkit, e.g., that of a plugin developer, or of a theorem prover developer who wants to use an external component as a decision procedure.

9.1 Selection of suitable components

The first thing to do when planning an integration is to select the theorem provers, model checkers, etc., to provide the required proof support in the application. There will almost certainly be trade-offs to be made in this choice. Issues to consider include:

- The logic of the proof engine: Is it expressive enough and well-suited to the application?
- Do the systems under consideration have compatible semantics?
- Can the systems be used as components, i.e., do they have a machine-oriented interface?
- Do the systems already have PROSPER interfaces?

A typical trade-off would be to select the second best decision procedure for some task because it already has a PROSPER interface, while the best one does not.

Some potential components may themselves be complex systems designed to be used directly by a human. The complexity need not be a problem in itself. The key question is whether a clear machine-oriented interface can be identified that will allow the tool to be used as a server component in a larger system.

If the application supports some formal language \( L \), it is likely that a formal semantics for \( L \) will be required in the proof engine. One issue then would be the suitability of a theorem prover’s logic and infrastructure for representing that semantics. If there is already support for \( L \) in some theorem prover, that would weigh heavily in its favour. This issue is discussed further in Sect. 9.3. The question of semantic compatibility of systems is beyond the scope of this paper.

9.2 Choice of proof engine

In practice, the choice of theorem prover to use as proof engine is likely to be quite limited, especially if PROSPER is to be used. PROSPER really assumes that the kernel of HOL98 will be used as the basis of the proof engine, though with some work, other systems might be used in its place. The most likely candidate is Isabelle/HOL [58] since it has a very similar logic and uses the same implementation language, Standard ML. HOL98 uses the Moscow ML implementation of Standard ML. If Isabelle is being used with a different implementation, there is likely to be some programming to do concerned with implementation-specific (system-level) features.

It would be significantly more difficult to use PVS [63], another popular theorem prover, as the proof engine. Its logic is more expressive than HOL’s and, consequently, the PROSPER Integration Interface does not support all the possible logical terms in PVS, at least not without some encoding. If the application did not require the full expressive power of the PVS logic, there might not be a problem. Just as importantly, though, PVS is implemented in Common Lisp for which there is currently no implementation of the PROSPER Integration Interface. See below for a discussion of implementation languages and related issues. It should also be noted that PVS is itself considered a means of combining verification procedures:

> More recently, we have come to view PVS as a semantic framework for integrating powerful deductive procedures. [64, pp. 7–8]

9.3 Support for the language of the application

In order to effectively conceal the underlying proof technology from the user of the application, it is usually necessary to provide some kind of translation from the language(s) of the application into higher order logic. To obtain confidence in the soundness of the translation, it is common to embed the semantics of the language(s) in the proof engine. Hence, the availability of an existing embedding, or the ease with which one can be created, is another important issue in deciding whether to use the PROSPER toolkit, and if so, how.

There is a wealth of literature on embedding all kinds of languages and formalisms in the higher order logic used by the HOL system. In fact, the activity has become sufficiently routine that tools have been developed to assist [13, 14, 61]. The embedded languages and formalisms include programming languages (e.g., [55]), hardware description languages (e.g., [16, 60]), specification languages (e.g., [18]), process calculi (e.g., [52]), and programming logics such as Hoare Logic (e.g., [35]).

Embeddings in HOL range from the straightforward, e.g., Linear Temporal Logic (LTL), to ones where the size and complexity, or the lack of a good semantic match, make it infeasible to implement a faithful embedding.

For example, languages that heavily exploit the untyped nature of set theory or make essential use of partial functions for expressive conciseness are not readily embedded in HOL. The embedding of a language whose semantics uses a fairly sophisticated mathematical domain (for example, untyped \( \lambda \)-calculus) may require a lot of work to develop that body of mathematics in HOL.

In some cases it may therefore be a lot easier to provide an embedding in a theorem prover that uses a significantly different logic, or perhaps an embedding already
exists in another system. This could outweigh all other considerations, but as can be seen from the range of embeddings referenced above, there is a sizeable library of existing embeddings in HOL.

The issue of embedded formalisms may also arise in connecting a plugin. It may be necessary to have an embedding of the plugin’s logic in the proof engine in order for the two components to interact soundly. The languages used by plugins are, however, usually quite straightforward to embed in HOL.

9.4 Using tools and procedures as PROSPER plugins

Considering now more general components, i.e., those that will be used as plugins, the first point to consider is whether the potential component already has a PROSPER interface. If it does, it should be easy to integrate. If not, the best option would be to ask the developers of the tool to add one, or collaborate with them to add one. If that is not possible, the next choice point is governed by the availability of source code. Without access to the source code, it is usually impossible to add a PROSPER interface. The PROSPER toolkit does, however, include a “harness” which can be used to wrap a black-box tool and present a PROSPER-compatible interface to the outside world.

The harness works by interacting with the input and output streams of the tool, converting them to string-based PROSPER interface data. The PROSPER developers strongly discourage this approach to integration, unless absolutely necessary, because the resulting interfaces tend to be untidy, lack robustness, and perform a lot of unnecessary parsing and unparsing. The job of presenting the plugin’s functions as a natural interface inside the proof engine ceases to be a simple transliteration process. The advantages of differentiation of data by type, provided by the PII, are also lost.

The Stanford Validity Checker (SVC) is an example of a tool for which source code is available and which has been successfully given a PROSPER interface without resorting to use of the harness (see Sect. 8.2). It helped enormously that SVC already had a fairly distinct and identifiable API – tools whose functionality is offered through a well-documented and coherent set of entry points are the most suitable for use as PROSPER plugins. Other examples of such tools include the Prover Plug-In [65, 67] and NuSMV [22].

9.5 Availability of a PII implementation

Assuming that source code is available, the next question is whether there is an implementation of the PII in the language of the component. Currently, there are PII implementations in Java (client-side only), C, and Standard ML. The C implementation can also readily be used in C++. In the absence of a compatible PII implementation, there are the following choices:

– Implement the PROSPER Integration Interface in the language of the component. There is a guide to assist in doing this [15], but it is a more involved task than normal use of the PROSPER toolkit for integration, and requires a lot of programming work. It also has the disadvantage that if the internals of the PII in the toolkit change, the custom implementation will also need to be updated.

– Use the C implementation of the PII via a foreign-function interface in the target language, if it has one. This is a lighter-weight approach than a full implementation but it suffers from some of the same disadvantages, and has the added disadvantage that the C implementation may not handle issues such as automatic memory management as well as a dedicated implementation for the language.

– Use the harness (see above).

The availability of a PII implementation for a given language is also an issue for the language in which the application program is written. On the whole, though, there are likely to be less problems because the user of the PROSPER toolkit should have access to the source code of the application.

9.6 Extension of the PII

The PROSPER Integration Interface supports a rich variety of data including dedicated support for logical terms. In the event that some data required by the application cannot be satisfactorily encoded using the PII, a user might consider extending the PII. This would, however, be a drastic step, since it would render the user’s system incompatible with other PROSPER components. In effect, it would produce a variant of the PROSPER toolkit. Although, pragmatically, this might be the best approach, it is not hard to see that it would undermine one of the key objectives of PROSPER, namely standardisation.

9.7 Possible architectures

Although the typical configuration of a PROSPER-based system is an application connected to one proof engine, which itself may be connected to zero or more plugins, other architectures are possible. There may be a database component that mirrors the theory data of the proof engine and can be accessed by the application and plugins (see Sect. 5). More radically, an application could use multiple proof engines. In addition, because the same interface is used for the application/proof-engine connections and the proof-engine/plugin connections, i.e., it is a generic client/server interface, longer chains of components are possible. For example, a plugin could itself use other components as servers.

One current restriction is that each server can have at most one client connected to it at any time. The database component may have multiple clients, but each client is
forced to disconnect between operations, so in effect the database does not have multiple simultaneous clients.

The PROSPER toolkit could also be used to connect an interactive theorem prover to other components; there does not have to be a separate application.

9.8 To use or not to use?

The preceding sections have explored practical issues in using the PROSPER toolkit to provide embedded proof support in an application. These factors should be considered when deciding whether to use PROSPER. The PROSPER toolkit consists of a significant body of software, so in that sense it is not a lightweight solution; but it is intended to make it easy to integrate proof support, as exemplified by the Microsoft Excel example described in Sect. 8.1, which took just a few days to complete. It is hard to imagine an ad hoc approach being any quicker. There is no avoiding some of the low-level integration work that PROSPER provides essentially for free.

No doubt, in some cases an ad hoc integration will be a quicker solution, but even then there are other advantages to taking a more principled approach, including:

- **Ease of maintenance**: Once a PROSPER-based solution has been constructed, it is likely to be much easier to maintain, modify, and extend it, than for an ad hoc solution.

- **Extra features**: Using PROSPER provides some important extra features. For example, it has been our experience that people considering integration of proof support do not, in the early stages at least, think about the need to interrupt the proof procedures. Interruptability is important because the run times of proof procedures can be highly unpredictable; in fact, some proof procedures are not guaranteed to terminate. Interrupts are a challenging technical issue, one which PROSPER goes a long way to solving for the integrator.

- **Reuse**: The PROSPER components can be used again later. This may be more of a communal advantage than a personal one. As with most new technology, early adopters will pay a higher price, but in so doing will make the technology cheaper for later adopters.

The key work that will almost certainly be required when using PROSPER is the construction of a custom proof engine. Although this requires programming in ML, the amount and difficulty of the programming may be quite small. In some cases the ML programming will be little more than writing code to “glue” together calls to existing proof procedures and plugins. Similarly, although PROSPER provides the expressive power of higher order logic, many applications will require knowledge of only the first-order subset, which should be familiar to most computer scientists. An integrator need only “buy into” the additional features of higher order logic if they are needed.

With regard to plugins, there has been a move recently amongst tool developers towards providing APIs. The presence of an API significantly simplifies the process of adding a PROSPER interface.

10 Related work

**Combined tools.** There has been a great deal of recent work in combining decision procedures (in particular model checkers) with theorem proving to increase the level of automation and the size of designs that can be dealt with by formal verification. Early experiments include links to model-checking based on embeddings of the modal mu-calculus in the logics of the HOL [2] and PVS [59] theorem provers.

Another notable effort was the HOL-Voss System [45], which provided a link to the Symbolic Trajectory Evaluation (STE [62]) model-checking algorithm from within HOL. This allowed HOL to invoke STE as an external decision procedure.

In much of this early work, external decision procedures were integrated into theorem provers in a predominantly “black-box” fashion. Experience in using these tools has shown that a more “glass-box” integration provides a higher level of control, which seems very important for practical effectiveness. A prominent example of glass-box integration is the Forte system [1], which intemately combines STE model checking and theorem proving in a single framework. This has been used very effectively for industrial-scale formal hardware verification [57].

In PROSPER, both black-box and glass-box tool integrations are possible, with a focus on glass-box integrations. Achieving the intimacy of some of the work described above is not, however, currently possible using the PROSPER Integration Interface because of the large binary decision diagrams involved. Communicating these between components is too inefficient. The wider PROSPER project has, nevertheless, addressed this by tightly integrating a BDD package into the implementation of ML used for proof engines [38].

In a slightly different vein the HOL/CLAM project [17] linked HOL to CLAM [20], 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. As previously discussed in Sect. 7.3 some similar issues also arise when connecting a graphical user interface to a proof tool [11].

The related work described above 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 also provide a generic framework for the integration of tools.
MathWeb [30] is a framework for distributed mathematical services. These services are other reasoning systems (e.g., resolution theorem provers and computer algebra systems). Knowledge can be stored in a service, MBASE [31], for the sharing of theory information between other services. MathWeb works on the assumption that services will be distributed with many users making use of one service as appropriate. There is no concept of customisation of a subset of services to a particular application as there is in the Prosper project.

Ω-ANTS [10] is an agent-based approach to combining interactive and automated theorem provers. It uses a blackboard architecture and other agent-oriented features to provide flexible interaction between the components. Like MathWeb, Ω-ANTS is not oriented towards providing proof support in applications.

ILF [25] is another framework for integrating interactive and automated provers, with an emphasis on providing a good user interface to the automated provers. The provers may be distributed and work concurrently, and Prolog is used as a scripting language much as ML is used in Prosper. Another framework, TechS, enables automated provers for first-order logic to co-operate by exchanging logical information [32]. So-called “referees” are used to filter the available data to avoid the provers being swamped.

Other related work, though not really an integration framework, is Ahrendt et al’s study of a tight integration of an automated theorem prover (3TAP) with an interactive proof system (KIV) in the domain of software verification [4]. They draw a number of conclusions from the exercise that they believe to be generally applicable when integrating these two kinds of system.

ETI [69], 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. 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 in which the PII is implemented to be treated as glass boxes. Tools that are not implemented in C++ may also be used with ETI provided they are encapsulated with C++ code where necessary.

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 [33], the control strategies used by the system [23], and the interaction mechanisms supported by the system [5]. 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 [46] and its predecessor Nqthm, so that they conform to the OMRS specifications. OMRS is supported by the Logic Broker Architecture [6] which is a CORBA-based system for the integration of reasoning systems specified in OMRS.

SAL (Symbolic Analysis Laboratory) is a recent collaborative effort that provides a framework for combining different tools to calculate properties of concurrent systems. SAL includes a language for specifying concurrent systems in a compositional way. Tools for abstraction, program analysis, theorem proving and model checking may be combined in the framework. One instance that has already been put together [9] involves the PVS theorem prover as a major component.

These systems all allow the integration and combination of verification components. Components are treated in a range of ways from a black-box approach to a glass-box approach. 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. However, we do not ignore logical issues and 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. While HLL allows some customising of tools into larger systems there is less of a focus on this aspect than there is in Prosper.

Design tools with embedded verification. There is a general trend towards integration of formal verification tools into the software and hardware design processes. For example, Braun et al. argue that for formal techniques to be useful they must be integrated into the design process [19]. The integration they describe goes beyond integration of tools to also discuss integration of semantics and integration into the methods of the development process.

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 [48] 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. However, the UniForM Workbench is a design tool in its own right.
rather than a toolkit for embedding verification into design tools. The Workbench also treats plugin decision procedures as black boxes.

Another software development tool using embedded verification is Extended Static Checking (ESC) [28] from Compaq Systems Research Center. ESC uses cooperating decision procedure technology first developed in the early 1980s to analyse Java programs for static errors. The emphasis is on using the formal verification technology to find flaws in the code rather than to perform a complete verification.

The KeY project aims to bridge the gap between object-oriented software engineering methods/tools and deductive verification. The KeY system integrates a commercial CASE tool with an interactive verification system and automated deduction techniques. One of the aims is to have interactive and automated verification tasks integrated in a uniform framework. The target programming language is Java Card. In a recent paper about the project [3], the authors argue that, amongst other things, tools for formal software specification and verification must be integrated into industrial software engineering procedures, and that an industrial verification tool should allow for gradual verification so that even software engineers who have little experience of formal methods may see the benefit.

The InVeSt tool [8] integrates the PVS theorem prover with the SMV model checker and uses them to discharge verification conditions generated by another program that is external to the theorem prover. PVS is therefore being used somewhat like PROSPER’s notion of a proof engine but unlike PROSPER, which allows a custom proof engine to be constructed with only the required features, InVeSt has to use the whole of PVS.

We are not aware of any project, other than PROSPER, with the specific aim to support the integration of existing components with the view to producing an embeddable customised proof engine.

11 Current status and future directions

At the time of writing a prototype toolkit and core proof engine have been developed and some significant changes have been made based on feedback from the application case studies. Several plugins have also been implemented and other plugins are being worked on, both by ourselves and by outside parties. We believe that the central concepts of a customisable proof engine and a language independent specification for communication are vital steps to the widespread use of embedded verification.

One area of research is the need to control which parts of a problem the theorem prover attempts and which parts it leaves to external plugins. This includes issues of the conversion of the representation used by a proof engine (e.g., sets) into the representation used by a plugin (e.g., propositional logic) in efficient ways.

Although soundness is not guaranteed by the PROSPER toolkit, it is nevertheless a concern. More mechanisms and methodologies need to be developed to more systematically support sound (or at least sane) translation between the semantics used by one part of a system and the semantics used by another. At present, PROSPER just considers the soundness of translation on an ad-hoc basis. The OMRS project, mentioned above, has developed more systematic frameworks for treating soundness issues.

Many proof attempts result in failure due to limitations of the proof technology, faulty specifications or faulty conjectures. In PROSPER, we have focused on applications where automatic proof is possible but this leaves the other causes of failure to be dealt with. It is desirable to give the user of an application feedback on why the proof attempt failed and, if at all possible, this should be in a form with which the user is familiar. Thus, a goal for further work is better support for producing and presenting counter-examples.

Another direction for future development would be a move to a different transport mechanism. For example, the Extensible Markup Language (XML) might be suitable for communicating the PROSPER interface data. Although more verbose than the current

ad hoc solution used by the PROSPER toolkit, XML includes mechanisms for sharing repeated structures and there is a wealth of tools available for XML. Other researchers have already done experiments in related applications, e.g., [31, 74].

More radically, a standard component architecture such as CORBA [56] could be used for the lower layers of the PII, leaving only the application support layer unchanged. It would be interesting to see to what extent standard architectures support the specialised requirements of verification-tool integration and, in particular, whether a lot of infrastructure would have to be built on top of the standard architecture.

A further question is whether the interface definition languages (IDLs) of CORBA, etc., could be used to express the PROSPER Integration Interface. The PII specification currently exists as a conventional document but not as a machine-readable interface specification. It seems likely that the PII could be expressed in some kind of IDL but we do not know whether existing IDLs would be suitable. It would also be interesting to try to generate PROSPER’s “language-specific bindings” (Sect. 6.1) from an IDL.

12 Conclusions

For (possibly invisible) embedded verification engines to gain widespread acceptance and use, verification tools must be customisable and easily combined. We believe the way forward draws on many of the standard aspects of component technology but also requires dedicated sup-
port, 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 case 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.

Acknowledgements. The authors thank Mike Gordon, Dirk Hoffmann, Graham Robinson, Alan Stevenson, and Kim Sunesen for their contribution to the work described in this paper, and Fatima Beltagui, Myra VanInwegen, and the anonymous referees for their comments.

References

5. Armando, A., Kohlhase, M., Ranise, S.: Communication protocols for mathematical services based on KQML and OMRS. In: Kerber and Kohlhase [47]
15. Boulton, R.J.: PROSPER Integration Interface implementor’s guide. ESPRIT LTR Project PROSPER (26241), part of deliverable D3.5, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, G12 8QQ, UK, April 2001
27. Dennis, L., Collins, G., Boulton, R., Robinson, G., Norrish, M., Slind, K.: The PROSPER toolkit version 1.4. ESPRIT LTR Project PROSPER (26241), part of deliverable D3.5, Department of Computing Science, University of


