Plug-in proof support for formal development environments

David Hemer[⋆] **Gregory Long**[†] **Paul Strooper**[†]

[⋆]School of Computer Science
The University of Adelaide, SA, Australia, 5005
Email: hemer@cs.adelaide.edu.au
[†]School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, Australia, 4072
Email: {gregl,pstroop}@itee.uq.edu.au

Abstract

A number of industrial software development standards mandate that safety-critical software components be developed using formal methods, including formal verification. While formal development is supported by a number of formal development environments, verification of correctness properties is still a major bottleneck. Most formal development environments provide built-in facilities for discharging these correctness properties (so-called proof obligations). However these built-in tools are typically less mature and sophisticated than stand-alone theorem provers. FDEs would benefit from being able to use a variety of theorem provers to discharge proof obligations, where different provers can be selected for different problem domains.

In this paper we describe a generic framework that supports the many-to-many connection of formal development environments and theorem provers. Before developing the framework we completed three case studies in order to reveal the main translation issues that need to be addressed. These translation issues were used as input to the requirements for our translation framework. We describe one of these case studies in detail in this paper. We then describe the framework and an Intermediate Modelling Language (IML), which is used to connect the FDEs to the theorem provers. The framework is supported by a collection of translators, both from FDEs (B and CARE) to the IML, and from the IML to theorem provers (Isabelle/HOL, Ergo and Otter).

1 Introduction

Developing correct software is important. Faults in software may lead to financial loss, harm to the environment, litigation, or even loss of human life. In cases where there is significant risk, software of high integrity is required. This can be achieved through the use of a rigorous development method.

Formal development environments (FDEs) (Elmstrom, Larsen & Lassen 1994, Craigen, Kromodimoeljo, Meisels, Pase & Saaltink 1991, Smith 1990, Lindsay & Hemer 1996, Abrial 1996) provide a methodology and tool support for producing high integrity software. Two key phases of the development process are: formally specifying the scope and behaviour of a software system, and "refining" this specification to an executable program. During both of these phases, properties may be generated that must be proven. These properties, often called proof obligations, can help demonstrate the correctness and consistency of a specification or refinement. The nature of these proof obligations means that it is unreasonable to expect humans to check them by hand; instead some mechanical support is required.

While most FDEs provide built-in support for mechanically checking proof obligations, this support is usually far less mature than the support offered by stand-alone theorem provers. Therefore, in order to discharge proof obligations, it would be beneficial if we were able to plug stand-alone theorem provers into FDEs. Furthermore we would like to be able to use different provers within the same FDE depending on the application domain (for example we might have one prover that is good at inequality reasoning and another that provides good support for inductive proofs; we would then select the particular prover depending on the application).

To be able to link multiple FDEs and theorem provers we first examined the main issues that are involved in translating between FDE and theorem prover notations. To do this we first completed three case studies involving translation of proof obligations from FDEs and into theorem prover notation. These case studies were a stack data structure and a square root approximation problem in B (Abrial 1996) and square root approximation in CARE (Lindsay & Hemer 1996). We describe the stack case study in detail in this paper.

For each case study, proof obligations were generated in the FDE. For the B examples, we attempted to discharge these proof obligations using the B toolkit's theorem provers; several of these proof obligations could not be proven using the B tools. For the CARE example, no proof was done prior to translation because the current version of the CARE tools does not provide any built-in prover support. The proof obligations were then translated by hand to Isabelle/HOL notation and then proven within Isabelle/HOL (Nipkow, Paulson & Wenzel 2002). All proof obligations were proven within Isabelle/HOL.

From the case studies and a review of the literature (Section 2), we identified a number of translation issues (Section 4). Our solution to the problem of translating between FDEs and theorem provers is a generic framework, based on the many-to-many architecture shown in Fig. 1. Our approach is based on translating via an intermediate language, rather than translating directly from FDE to theorem prover.

The framework consists of an intermediate modelling language (IML), tables of supported constructs, and guidelines for developing translators (FDE to IML and IML to theorem prover). Our solution to translation is syntax-based, in that the IML and supported constructs are represented at a syntactic level, rather than at a semantic level, and transla-

David Hemer was previously in the School of Information Technology and Electrical Engineering, The University of Queensland.

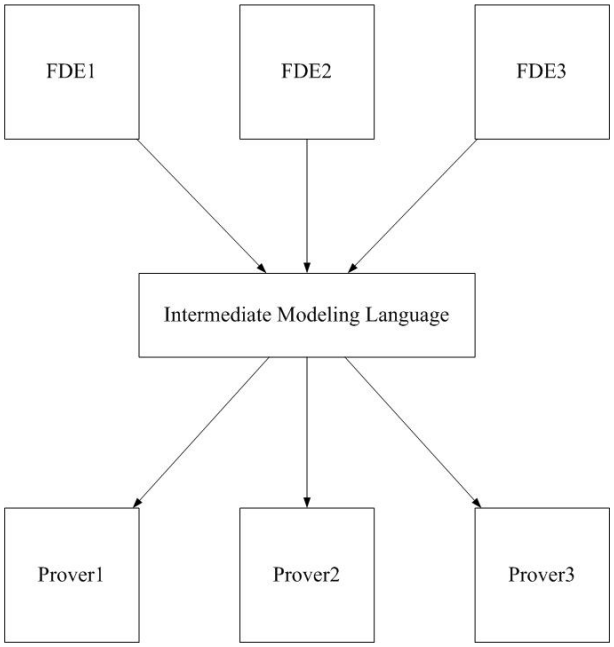

Figure 1: Translation framework

tion is syntax-based. The obvious disadvantage is that we cannot check the soundness or consistency of translation. However our syntax-based approach is lightweight and has the practical advantage that we have been able to enhance the proof support for B and provide proof support for CARE. The framework is supported by a collection of translators. At present we have implemented prototype support for the B and CARE FDEs and the Isabelle/HOL, Ergo (Utting, Robinson & Nickson 2002) and Otter (Kalman 2001) theorem provers.

The main benefit of this approach is that the conceptual gap between the FDE and translation medium is less than the gap between the FDE and theorem prover; therefore it is easier to add a new FDE to the system, while maintaining existing links to theorem provers. Likewise, the conceptual gap between the translation medium and theorem prover is less than the gap between FDE and prover; therefore it is easier to add new theorem provers to the system, which can then interface with all FDEs that are linked to the system.

Section 2 contains an overview of related work. Section 3 describes the stack case study in the B-Toolkit, comparing the proof capabilities of the B-Toolkit against Isabelle/HOL and describing the main issues that we encountered in translating proof obligations from B notation into Isabelle/HOL notation. Section 4 describes the translation issues that were revealed during the three case studies and the literature review. In Section 5 we describe the generic framework. Section 6 contains a discussion of outstanding translation issues.

2 Related Work

To evaluate the feasibility and requirements of the framework, we have selected a representative sample of available formal development environments and theorem provers. It is not our aim to study all available development environments and provers, merely a diverse subset. We also discuss related work in the translation of logics, because this gives us an insight into the underlying challenges that need to be addressed in translating between FDEs and theorem provers.

2.1 Formal Development Environments

Formal development environments, such as *The B-Toolkit* (Abrial 1996), *IFAD VDM-SL Toolkit* (Elmstrom et al. 1994) and *CARE* (Lindsay & Hemer 1996), support the specification, design, implementation and maintenance of high integrity software systems. Each tool has its own formal mathematical logic in which specifications and refinements are written. For example, The B-Toolkit's specification language is built on the *Abstract Machine Notation*, an extension of Dijkstra's guarded command language (Dijkstra 1976). Each tool's logic can vary enormously with differing type systems, language richness, and capabilities.

Some formal development environments also have facilities to verify the correctness of a development. Proof obligations may be generated from specifications and refinements that can be discharged to ensure correctness.

2.2 Theorem Provers

A theorem prover can be used to prove proof obligations generated during formal developments. Some formal development environments (e.g. The B-Toolkit) incorporate proof tools. Unfortunately these built-in proof tools are often less mature than standalone proof tools. Interactive theorem provers such as *Isabelle/HOL* (Nipkow et al. 2002), *Ergo* (Utting et al. 2002), and *PVS* (Owre, Rushby & Shankar 1992) are more mature, have well-populated theory bases, and provide powerful tactic support. OTTER (Kalman 2001) is the most widely used automated theorem prover, and has been applied to a wide variety of problems. This generally makes them more suitable for discharging complex proof obligations than proof tools provided with formal development environments.

At present we have linked three theorem provers to the framework (Isabelle/HOL, Ergo and Otter). For the purpose of brevity we will focus on the Isabelle/HOL prover in this paper. *Isabelle* (Paulson 1989) is an interactive generic theorem prover built on a higher-order meta logic written in *Standard ML*. This meta-logic allows users to instantiate their own logics for reasoning. A variety of logics are implemented in the Isabelle environment, but we focus on the embedding of higher-order logic (Isabelle/HOL) because it is mature, has good tactic support, and is familiar to the authors. The syntax of Isabelle/HOL is based on the λ-calculus and functional programming. Object-level types are associated with meta-level types to take advantage of Isabelle's built-in type checker. Isabelle/HOL also identifies object-level functions with meta-level functions using Isabelle's operations for abstraction and application. Isabelle offers comprehensive support for automating proof in the form of tactics and tacticals (functions that combine tactics).

2.3 Translation issues

There are numerous case studies demonstrating translations between logics, embeddings of one logic in another, or incorporating prover support into a tool (Naumov, Stehr & Meseguer 2001, Agerholm 1996, Stringer-Calvert, Stepney & Wand 1997, Bodeveix & Filali 2002, Chartier 1998, Ahrendt, Baar, Beckert, Giese, Hahnle, Menzel, Mostowski & Schmitt 2002). Some of the issues raised in these papers are discussed below.

Naumov et al. (2001) describe problems encountered building a proof translator from HOL to NuPRL. HOL inference rules, when taken literally, are unsound in NuPRL due to the difference in typing systems. HOL has a built-in type-checking mechanism that guarantees a HOL term is well-formed. NuPRL has a weaker type-system that allows any syntacticly valid expression as a term, not guaranteeing its well-formedness. Well-formedness is enforced during a NuPRL proof, something that does not occur in HOL. HOL also has no notion of proof objects (recordings of proofs) to export into an external prover. Naumov was forced to add proof-recording facilities to HOL in order to extract proofs. In some cases, finding NuPRL equivalents to HOL constants was difficult so Naumov introduces explicit NuPRL counterparts.

Agerholm (1996) looks at translating specifications from VDM-SL (Jones 1990) to PVS. VDM-SL is represented as a shallow embedding in PVS, i.e., the syntax of VDM-SL constructs is not represented in PVS, instead they work with the "semantics" directly. Agerholm states that translation is not logically safe and notes that this is a problem of all shallow embeddings (Boulton, Gordon, Gordon, Herbert & Tassel 1992).

Stringer-Calvert et al. (1997) take Z theorems and prove them in PVS, but the built-in tactics were not well-suited to their requirements, so development of their own tactics was needed. Other challenges are noted, for example, Z specifications allow partial functions but PVS does not, requiring that partial functions had to be translated to total functions.

The papers discussed above all deal with one-to-one translations, however we are interested in many-to-many translations. Two projects, OMRS and PROSPER, attempt to deal with the problem of many-to-many translations.

The Open Mechanised Reasoning Systems (OMRS) project (Giunchiglia, Bertoli & Coglio 1998) is aimed at providing a framework for specifying, structuring, and inter-operating provers. The OMRS project is motivated by the fact that it is difficult to combine current provers due to their stand-alone nature and inadequately defined interfaces. For each prover the framework defines: the logic of the prover; how theorems are proved; and the interface of the prover. However OMRS is aimed at the development of new provers, whereas our aim is to connect existing provers. Connecting existing provers to the OMRS would require significant re-engineering of these provers.

The Proof and Specification Assisted Design Environments (PROSPER) toolkit (Dennis, Collins, Norrish, Boulton, Slind, Robinson, Gordon & Melham 2000), is focused on allowing provers (and other verification tools) to be interfaced with CAD/CASE tools in a modular way. A goal of the project is to develop a common theorem-proving infrastructure, based on HOL98. This proof support will be integrated into two CAD/CASE platforms, software verification via VDM-SL and hardware verification via a verification workbench that supports the CAD languages Verilog and VHDL. New verification tools (for example, a prover or model checker) can be incorporated as a plug-in to the core verification tool by extending an API that interfaces with HOL98.

MathWeb (Franke & Kohlhase 1999) is a system that allows existing, stand-alone theorem provers to be connected in an integrated, networked proof environment. This proof environment gains the services from integrated modules, and each module gains from using the features of other, integrated components. MathWeb provides the functionality to encapsulate proof assistants as objects providing mathematical services to a system bus, the system's communication mechanism.

Modules are autonomous agents that communicate using an XML-based language, KQML (Finin, Fritzson, McKay & McEntire 1994), with mathematics and proofs represented in OpenProof (Franke, Hess, Jung, Kohlhase & Sorge 1999), an extension of the OpenMath standard. Semantics are captured in standard content dictionaries, each mathematical system implements transformation procedures, known as phrase books, that interpret OpenMath representations into representations of the mathematical system. This mechanism forms the mathematical link between each module and MathWeb.

MathWeb differs from our approach in that they are concerned only with connecting theorem provers, whereas we are concerned with connecting theorem provers to FDEs.

We finish by looking at two papers that discuss issues relating to translation between logical systems. Saaltink, Craigen, Kromodimoeljo & Pase (1992) discuss the problems with the exchange of information (libraries and theories) between different theorem provers. They identify three major problems in achieving a sound translation of libraries:

1. It is difficult to translate material while preserving its meaning. Even if this exchange is possible the translated material may appear so unnatural (in order to preserve its meaning), that it is not usable.

2. The presentation of a theory is affected by the notation used and the capabilities of the proof tool, even if two tools have similar semantic bases. The representation of a rule can affect how easily it can be used within a proof.

3. Tools differ in the handling of definitions and some lack facilities for defining or extending libraries.

Watson (2001) describes a generic proof checker that reads and checks the proofs produced by a theorem prover. Watson raises a number of translation issues that were discovered while developing a generic proof representation. Specific issues include differing approaches to typing (strongly-typed versus untyped) and the representation of not-free-in constraints in rules.

3 Case study

This section outlines the specification and refinement of a stack data structure using the B method and toolkit (Schneider 2001). The B method and toolkit support the refinement of high-level formal specifications to implementable code. Specifications, refinements and implementations are represented as abstract machines, which are defined using a formal notation. Specifications introduce the state variables and the operations of the program. Refinements refine these state variables and operations so that they are closer to an implementable form, whilst maintaining the meaning of the original specification. Implementations represent the final stage in the refinement process, providing implementations for all of the operations in terms of an implementable state variable representation. B generates proof obligations that establish the correctness of the abstract machines. The toolkit includes both an automated theorem prover and an interactive prover that are used to discharge proof obligations.

Proof obligations that verify the correctness of the specification and refinement of the stack data struc-

ture were generated by the B toolkit. The proof obligations were translated by hand into Isabelle/HOL theorems, which were then proven. The aim was to gain insight into the issues associated with translating between an FDE and a theorem prover. Two other case studies (one in CARE and another in B) were also completed before we compiled the requirements for the framework. We discuss, in detail, the specification and refinement machines below with proof obligations generated from both.

3.1 Specification Machine

The *Stack* specification machine, shown in Figure 2, specifies a data structure for representing a stack, together with operations that manipulate it. The *Stack* machine defines a constant *max* that limits the maximum size of the stack, and specifies that *max* is an element of the set of natural numbers and that its value is ten. The state of the stack is stored in the variable *stack*. An invariant on the state specifies that the type of the *stack* variable is a sequence of natural numbers and imposes a maximum size on it.

MACHINE Stack
CONSTANTS max
PROPERTIES $max \in \mathbb{N} \wedge max = 10$
VARIABLES $stack$
INVARIANT
$\text{size}(stack) \leq max \wedge stack \in \text{seq}\ (\mathbb{N})$
INITIALISATION
$stack := [\,]$
OPERATIONS
Push$(elm) \mathrel{\widehat{=}}$
PRE
$elm \in \mathbb{N} \wedge \text{size}(stack) < max$
THEN
$stack := stack \frown [elm]$
END ;
$out \longleftarrow$ **Pop** $\mathrel{\widehat{=}}$
PRE
$stack \neq [\,]$
THEN
$out := \text{last}(stack)$
$\parallel\ stack := \text{front}(stack)$
END
END

Figure 2: Stack Abstract Machine Specification

The stack is initialised to the empty sequence within the initialisation operation. The stack machine also includes specifications of two operations **Push** and **Pop**. The **Push** operation appends an element, *elm*, to the top of *stack*. The operation includes a precondition stating that *elm* must be member of the set of natural numbers and the size of *stack* must be less than *max*. Provided the stack is not empty, the operation **Pop** performs two substitutions simultaneously using parallel substitution, ($\parallel$), removing the last element of the stack and assigning it to the output variable *out*.

After specifying the stack machine, proof obligations that check the machine's consistency were generated by the B toolkit. Specification proof obligations check for, amongst other things, consistency of the invariant, consistency of the initialisation and other operations, and satisfiability of the constraints, properties and invariant (Schneider 2001). Seven proof obligations were generated at this stage; all were discharged automatically by the B toolkit's automated prover.

3.2 Refinement Machine

The specification machine *Stack* is refined to the refinement machine *StackR* (which is almost directly implementable) in Figure 3 with the sequence (*stack*) being refined to an array (*stackr*). The structure of *StackR* follows that of the specification except that we identify the specification machine (*Stack*) that this machine refines.

REFINEMENT $StackR$
REFINES $Stack$
VARIABLES $stackr, num\_elmsr$
INVARIANT
$stackr \in 0..max - 1 \rightarrow \mathbb{N} \wedge$
$num\_elmsr \in \mathbb{N} \wedge$
$num\_elmsr \leq max \wedge$
$num\_elmsr = \text{size}(stack) \wedge$
$\forall xx.(xx \in 0..num\_elmsr - 1$
$\Rightarrow stackr(xx) = stack(xx + 1))$
INITIALISATION
$num\_elmsr := 0;$
ANY ff **WHERE**
$ff \in 0..max - 1 \rightarrow \mathbb{N}$
THEN
$Stackr := ff$
END
OPERATIONS
Push $\mathrel{\widehat{=}}$
BEGIN
$stackr(num\_elmsr) := elm;$
$num\_elmsr := num\_elmsr + 1$
END;
$out \longleftarrow$ **Pop** $\mathrel{\widehat{=}}$
BEGIN
$num\_elmsr := num\_elmsr - 1;$
$out := stackr(num\_elmsr)$
END
END

Figure 3: Stack Abstract Machine Refinement

The data-refined stack state is represented by the variable *stackr*, to store the contents of the stack, and the variable *num_elmsr*, representing the number of elements currently in the stack. The first two lines of the invariant define the types of *stackr* and *num_elmsr*. The third line of the invariant states that the value of *num_elmsr* should not exceed *max*.

The final two lines of the invariant define a relationship between the specification and refinement data structures (often referred to as a coupling invariant). The fourth line states that the value of *num_elmsr* should equal the size of the stack. The specification data structure, *stack*, and the refinement data structure, *stackr*, are linked by a relationship stating that all the elements of *stack* occur in the same order in *stackr* (although *stack* is indexed from 1 and *stackr* is indexed from 0).

The operations from the *Stack* machine are refined by specifying them in terms of the concrete representation with the refined operations using sequential composition, (;), in place of parallel substitution. Any preconditions from specification operations are assumed for the corresponding refinement operation.

The initialisation operation sets *num_elmsr* to zero and non-deterministically assigns *stackr* to a value of type $0..max-1 \rightarrow \mathbb{N}$. The **Push** operation adds an element at the position *num_elmsr* in *stackr* and then increments *num_elmsr*. The **Pop** operation decrements *num_elmsr* and returns the element at the new position of *num_elmsr*.

Proof obligations are generated for the *StackR* refinement machine that check the machine's consistency and prove that *StackR* is a refinement of its specification, *Stack*. For example, proof obligations showing that the execution of the refined **Push** operation satisfies the invariant are generated for each predicate of the coupling invariant. One such proof obligation, *Push5*, is presented below.

$$
\begin{array}{l}
max \in \mathbb{N} \wedge max = 10 \wedge stackr \in 0 \mathinner{..} max - 1 \rightarrow \mathbb{N} \wedge \\
num\_elmsr \in \mathbb{N} \wedge \\
num\_elmsr \leq max \wedge num\_elmsr = \mathrm{size}(stack) \wedge \\
\forall\, xx \bullet (xx \in (0..num\_elmsr - 1) \\
\quad \Rightarrow stackr(xx) = stack(xx + 1)) \wedge elm \in \mathbb{N} \wedge \\
\quad \mathrm{size}(stack) < max \\
\qquad \Rightarrow xx \in (0..num\_elmsr + 1 - 1) \\
\qquad\quad \Rightarrow (stackr \oplus \{num\_elmsr \mapsto elm\})(xx) \\
\qquad\qquad = (stack \frown [elm])(xx + 1)
\end{array}
$$

It states that after the execution of **Push**, the properties and invariant clauses of the refinement machine and the precondition of **Push** imply that (with the new element added) each element of *stackr* corresponds to an element of *stack*.

The proof obligation *Pop4* for the **Pop** operation states that the final line of the invariant for *stackr* is maintained after the **Pop** operation is performed. The proof obligation assumes that the invariant holds before the operation and that the precondition of **Pop** holds.

$$
\begin{array}{l}
max \in \mathbb{N} \wedge max = 10 \wedge stackr \in 0 \mathinner{..} max - 1 \rightarrow \mathbb{N} \wedge \\
num\_elmsr \in \mathbb{N} \wedge \\
num\_elmsr \leq max \wedge num\_elmsr = \mathrm{size}(stack) \wedge \\
\forall\, xx \bullet (xx \in (0..num\_elmsr - 1) \\
\quad \Rightarrow stackr(xx) = stack(xx + 1)) \wedge stack \neq [\,] \\
\qquad \Rightarrow xx \in (0..num\_elmsr - 1 - 1) \\
\qquad\quad \Rightarrow stackr(xx) = \mathrm{front}(stack)(xx + 1)
\end{array}
$$

In total fourteen proof obligations associated with the correctness of the refinement machine were generated. Of these, twelve were discharged using the B proof support tools (B Autoprover and B-Tool prover). The remaining two proof obligations (*Push5* and *Pop4*) could not be proven due to limitations in the theory base of the B provers. While new rules can be introduced in the B-Tool prover, they can only be introduced as axioms. The soundness of these rules is not checked, nor are there any checks done to determine whether the consistency of the theory base is maintained. Clutterbuck, Bicarregui & Matthews (1996) try to solve this problem by proposing a method of representing derived rules as proof steps in B. Unfortunately even they admit that their process results in rules that are cumbersome to use.

3.3 Translation to Isabelle/HOL

All of the proof obligations associated with the stack case study in B were hand-translated to Isabelle/HOL theorems. The seven proof obligations associated with the specification machine were discharged automatically within Isabelle/HOL. All remaining proof obligations were interactively discharged in Isabelle/HOL. Several theory extensions were required in Isabelle/HOL, however these extensions were represented as lemmas that were subsequently proven, thus maintaining the soundness and consistency of the theory base.

Various challenges arose during the translation of B proof obligations to Isabelle/HOL. These issues are illustrated below. In Section 4 we discuss the translation issues more generally in the context of our framework.

The first challenge was representing B type constraints in Isabelle/HOL. In B the type constraints are represented explicitly in proof obligations as set memberships; for example, the condition $max \in \mathbb{N}$ in the proof obligation *Push5*. Since Isabelle/HOL is strongly typed and type checking is handled by the built-in type checker prior to proof, such set membership conditions can usually be avoided. However, for generality, we translate set membership conditions in B to the corresponding condition in Isabelle/HOL; for example, the condition $max \in \mathbb{N}$ is translated to (*max* is renamed to *bmax* to avoid name clashes):

$$bmax \in \{n :: \mathbb{N} \bullet True\}$$

where "::" is the Isabelle typing operator, which is different to the set membership operator $\in$.

While in this case it is not strictly necessary to include the type constraint, *True*, it is necessary when there is no direct correlation between the B type and the Isabelle type. For example, if the constraint was $max \in \{n : \mathbb{N} \mid n > 5\}$ instead, then it would be necessary to include the translated constraint (i.e. $max \in \{n :: \mathbb{N} \bullet 5 < n\}$).

A second problem arose when representing the type of *stackr* ($stackr \in 0 \mathinner{..} max - 1 \rightarrow \mathbb{N}$) in Isabelle/HOL. We cannot represent the type $0 \mathinner{..} max - 1 \rightarrow \mathbb{N}$ directly in Isabelle/HOL as total functions must have basic types for their domain and range. We can however represent the type as a partial function, $\mathbb{N} \nrightarrow \mathbb{N}$, with an extra constraint that defines the domain:

$$stackr \in \{t :: (\mathbb{N} \nrightarrow \mathbb{N}) \bullet \mathrm{dom}\, t = \{i \bullet 0 \leq i < bmax\}\}$$

Partial functions are modelled in Isabelle/HOL as total functions that map an element x to the value *Some* y, when the function is defined at x and where y is the value of the function at x, or to the constant *None* when the function is not defined at x. While this representation of partial functions is generally adequate, it does introduce some extra translation overheads. In particular, when assigning values to a partial function the *Some* and *None* labels must be included. Similarly, when accessing values from the partial function these same labels must be stripped off using the *the* operator. For example, the term $stackr(xx)$, which accesses the element at index xx of *stackr*, is translated to *the* $(stackr(xx))$.

The third translation problem was representing sequences in Isabelle/HOL. Isabelle/HOL does not provide support for a sequence data type, but does support a list data type. For our purposes these data structures are similar enough so that we can use lists instead of sequences. However, a difference between the two data structures is that lists in Isabelle/HOL are indexed from *zero* whereas sequences in B are indexed from *one*, therefore adjustments to the indices need to be made when dereferencing the list.

The final translation problem was avoiding name clashes with reserved words in Isabelle/HOL. For the stack proof obligations the variable *max* is renamed to *bmax* to avoid a name clash with the *max* reserved word in Isabelle/HOL.

4 Translation issues

In this section we summarise the main translation issues that were discovered during the case studies and

the literature review. These translation issues were used as inputs into the statement of the requirements for our generic framework.

Type representation: We have seen two different ways of representing type information: at the meta-level using built-in types, or at the object-level using set notations. While it is possible to translate between the two representations, we must be aware of the consequences. At the meta-level, type checking can be handled by built-in tools prior to proof, while at the object-level, type checking becomes part of the proof. Consequently, after translating from meta-level to object-level representations, proofs typically become more difficult. Type representation is much richer at the object-level than at the meta-level, so in the worst case we may lose expressiveness by translating from object-level type representations to meta-level representations; in the best case the translation leads to more cumbersome type representations.

Higher-order constructs: As well as wanting to discharge proof obligations associated with a particular program development, we may also want to discharge proof obligations associated with the correctness of reusable library components. Such components are often specified using higher-order logics in order to ensure that they are generic and are applicable in a variety of situations. Therefore to prove properties about such generic components we need to be able to represent higher-order constructs.

Fortunately there are a number of theorem provers that offer support for higher-order logics, and such provers would be obvious candidates. However, while higher-order logics are more expressive, they are also more complex. Indeed in most cases we may wish to stay within first-order logic. To do this we want some mechanism that allows us to switch to higher-order logic constructs only when required.

Construct support: In translating between an FDE and a theorem prover we need to determine counterparts for representing constructs in the FDE and theorem prover. As we have seen in the case study, direct counterparts are not always available; however there may be constructs that are similar and can be used if adapted in some manner. To use these similar constructs we must be able to determine the differences and how they can be adapted. This will be very difficult to do in general, but there are certain constructs that can be easily adapted. For example lists/sequences can be adapted by changing indices. Orderings can be adapted by reversing the arguments.

As Saaltink et al. (1992) point out, using adapted constructs can result in proof obligations that are more difficult to read and reason about.

Theory extensions: A number of FDEs, CARE being one example, let the user write domain-specific theories that include type and function definitions. In translating these theory extensions into the target theorem prover representation, we must ensure that these extensions maintain the consistency and soundness of the theory base. Many theorem provers provide facilities for making conservative extensions to theories; such extensions must adhere to a strict form. It is therefore important that the framework provides support for representing conservative extensions, rather than having to represent any extensions as axioms (as is done in B).

Another related issue is the representation of lemmas (or any other construct that has an associated proof), such as those that are used to prove a proof obligation. Being able to export these lemmas is useful if we want to discharge proof obligations in another theorem prover. However sharing lemmas is more difficult than sharing definitional extensions, because lemmas also have an associated proof. We can address this issue in a number of ways including:

1. Representing the lemma as an axiom and trusting the proof in the other theorem prover. However this may affect the soundness of the logic.
2. Representing the lemma as a lemma in the new theorem prover and redoing the proof in the new prover. This means that the soundness of the prover's logic is maintained, but we have the added burden of redoing the proof.
3. Representing the lemma as a lemma in the new theorem prover and importing the original proof using a generic proof representation (Watson 2001).

Undefinedness: A number of commonly used specification constructs are undefined for certain input values. One example is natural number division, which is undefined when the divisor is zero. Another example is the function *head* for accessing the head of a sequence, which is undefined when the sequence is empty.

Treatment of undefinedness varies across different FDEs and theorem provers. In some cases the issue of undefinedness is largely ignored, with constructs only defined for values in the domain. For example, in Isabelle/HOL, *head* is defined as a total function on lists, but is only defined for non-empty lists. The expression $head([\,])$ is well-formed in Isabelle/HOL, but cannot be simplified any further.

Other FDEs (e.g. VDM-SL) and provers (e.g. Ergo) represent undefinedness explicitly. Ergo includes an undefined value ($\perp$), enabling the user to check whether or not an expression is defined. In this case the expression $head([\,])$ would simplify to $\perp$, indicating that a function has been used incorrectly.

In translating from a logic that supports undefinedness to one that does not, there will be a loss of information. In contrast, it may be possible to translate from a logic with no support for undefinedness to one with support and actually gain the ability to check for undefinedness.

Partial functions: The use of partial functions in abstract specifications is quite common, however not all theorem provers provide support for partial functions, while others offer a work-around solution. Weakly-typed set-theory based logics provide the best support for modelling and reasoning about partial functions. In this case, the domain of the partial function can be modelled precisely using sets, and it is possible to reason about whether or not particular values are in the domain of the function. On the other hand, strongly-typed logics cannot accurately model partial functions. Some of these logics provide no support at all; total functions must be used instead. Others, such as Isabelle/HOL, base partial functions on total functions, but map values outside of the function domain to an arbitrary undefined value. Because provers offer different solutions to this problem we cannot deal with the problem in a systematic way, but we must be able to recognise partial functions.

Variable representation: Proof assistants and formal development environments differ in the kinds of variables that can be represented in their logics. Examples of variables include: first-order (object)

variables; higher-order (function) variables; and meta (term or predicate) variables. In translating between FDEs and theorem provers, we need to ensure that we are translating to the right kind of variable. So we need to recognise what kinds of variables we are using in the FDE and properly distinguish between the different kinds of variables; then we need to match these kinds of variables with variables in the target theorem prover where possible.

Name spaces: It is easy to introduce name clashes between variables and predefined constructs when translating. Therefore we need a systematic solution to naming of constructs to avoid name clashes. In renaming variables we need to be aware of any variable naming conventions associated with the target representation language.

5 Generic framework

The framework encompasses an intermediate modelling language (IML) and automated translators that connect formal development environments and theorem provers to the IML. Proof obligations from formal development environments can be translated into IML syntax and then translated to an appropriate theorem prover. Fig. 1 presents the general structure of the framework.

5.1 Intermediate modelling language

The framework uses an intermediate modelling language (IML) that allows the modelling of theorems from a wide variety of formal development environments and theorem provers. The IML is designed to be supported by automated translation support with a syntax that can be easily parsed. The syntax of the IML, specified in Z (Spivey 1992), is shown in Fig. 4.

IML specifications are modelled as a theory (*THEORY*). A theory consists of a sequence of theory components (*THEORYCOMPONENT*), representing theorems, definitional extensions and axioms. Proof obligations are represented as theorems, consisting of a name (*NAME*), a sequence of variable declarations (*VARDECL*), identifying any free variables in the theorem, and a formula representing the theorem itself (*FMLA*). Variable declarations associate a variable with a set (representing the set of values that the variable can take), and a sort. Axioms have the same structure as theorems; we distinguish between the two since we want to ensure that they can be treated differently by the prover.

To support conservative theory extensions, the IML supports two kinds of definitional extensions: functional definitions and relational definitions. A functional definition consists of: a function name; a sequence of free variables appearing in the definition; a signature for the function, modelled as a set; and the body of the definition, modelled as an expression. A relational extension is defined in a similar manner.

Related theory components can be grouped together in a collection. A *defcollect* allows related theory components to be grouped as collections. It consists of a label (*NAME*) and a sequence of theory components that are related. For example, one can group a collection of definitions that constitute a primitive recursive definitional extension.

The IML models standard logical formulae including: the constants *true* and *false*; negation; the binary connectives conjunction, disjunction, implication and equivalence; and universal, existential and unique existential quantifiers. Also included is a construct for representing relation applications that take a sequence of arguments and evaluate to a boolean value. A relation is represented by giving its name, a sort, and a sequence of arguments.

NAME	***SORT***	**Description**
$=$	$X \times X \rightarrow \mathbb{B}$	Expression Equivalence
$\in$	$X \times X \rightarrow \mathbb{B}$	Set membership
$<$	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$	Less Than
$\leq$	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$	Less Than Or Equal To

Table 1: Relation constructs

NAME	***SORT***	**Description**
Natural Numbers		
plus	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$	Addition
minus	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$	Subtraction
mult	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$	Multiplication
divide	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$	Division
Sets		
size	$\mathbb{F}\, X \rightarrow \mathbb{N}$	Cardinality
union	$\mathbb{F}\, X \times \mathbb{F}\, X \rightarrow \mathbb{F}\, X$	Union
diff	$\mathbb{F}\, X \times \mathbb{F}\, X \rightarrow \mathbb{F}\, X$	Difference
intersect	$\mathbb{F}\, X \times \mathbb{F}\, X \rightarrow \mathbb{F}\, X$	Intersection
Sequences		
seqref	$\mathrm{seq}\, X \times \mathbb{N} \rightarrow X$	Dereference
front	$\mathrm{seq}\, X \rightarrow \mathrm{seq}\, X$	Front
append	$\mathrm{seq}\, X \times \mathrm{seq}\, X \rightarrow \mathrm{seq}\, X$	Append
length	$\mathrm{seq}\, X \rightarrow \mathbb{N}$	Length
last	$\mathrm{seq}\, X \rightarrow X$	Last
Functions		
fover	$(X \nrightarrow Y) \times (X \nrightarrow Y) \rightarrow (X \nrightarrow Y)$	Function Override

Table 2: Function constructs

The table of relation constructs (Table 1) presents a partial listing of available relations. This is only an initial list that will be extended as the tools are developed further. The table provides a *NAME* and *SORT* for each relation and a description of the relation's behaviour. Use of the *rel* construct involves the selection of a relation from the table and inserting its *NAME* and *SORT* in the appropriate place.

IML expressions (*EXPR*) are partitioned into first-order (*FOEXPR*) and higher-order (*HOEXPR*) expressions.

First-order expressions can be either function applications, variables, collections, or ordered pairs. Function applications are similar to relation applications, consisting of a function name, a sort, and a sequence of arguments. Table 2 gives a partial list of supported functions. Each function in the table is given a name, a sort and a description. We explain how this table is used to develop translators in Section 5.2.

Higher-order expressions are represented by the *HOEXPR* construct. Higher-order function application is represented by the *hofunapp* consisting of a *HOAPPLIC* node, either a higher-order functor or an expression, and a sequence of *EXPR* parameters. The *lambda* construct defines an anonymous function and *hopair* defines a pair of possibly higher-order variables. Higher-order variables are represented by the *hovar* construct with a *NAME* and *SORT*.

Although it is not necessary to provide a sort for each variable it makes the development of translators easier and provides a mechanism for redundancy checking. Sorts attached to variables can be compared to the sort of the construct of which it is an argument or compared to the sort specified at its declaration (a *VARDECL* construct). The *hovar* and *var* constructs are different to variable declarations because they refer to the use of a variable rather than its declaration.

$[NAME]$

$THEORY == \text{seq}\ THEORYCOMPONENT$

$$
\begin{array}{lcl}
THEORYCOMPONENT & ::= & theorem\langle\langle NAME \times \text{seq}\ VARDECL \times FMLA\rangle\rangle \\
& | & axiom\langle\langle NAME \times \text{seq}\ VARDECL \times FMLA\rangle\rangle \\
& | & edef\langle\langle NAME \times \text{seq}\ VARDECL \times SET \times EXPR\rangle\rangle \\
& | & pdef\langle\langle NAME \times \text{seq}\ VARDECL \times SET \times FMLA\rangle\rangle \\
& | & defcollect\langle\langle NAME \times \text{seq}\ THEORYCOMPONENT\rangle\rangle
\end{array}
$$

$$
\begin{array}{l}
VARDECL == NAME \times SET \times SORT \\
\begin{array}{lcl}
FMLA & ::= & true \\
& | & false \\
& | & not\langle\langle FMLA\rangle\rangle \\
& | & and\langle\langle FMLA \times FMLA\rangle\rangle \\
& | & or\langle\langle FMLA \times FMLA\rangle\rangle \\
& | & implies\langle\langle FMLA \times FMLA\rangle\rangle \\
& | & iff\langle\langle FMLA \times FMLA\rangle\rangle \\
& | & quant\langle\langle QUANT \times VARDECL \times FMLA\rangle\rangle \\
& | & rel\langle\langle NAME \times SORT \times \text{seq}\ EXPR\rangle\rangle \\
QUANT & ::= & forall \mid exists \mid uexists \\
EXPR & ::= & foexpr\langle\langle FOEXPR\rangle\rangle \\
& | & hoexpr\langle\langle HOEXPR\rangle\rangle \\
FOEXPR & ::= & funapp\langle\langle NAME \times SORT \times \text{seq}\ FOEXPR\rangle\rangle \\
& | & var\langle\langle NAME \times SORT\rangle\rangle \\
& | & pair\langle\langle FOEXPR \times FOEXPR\rangle\rangle \\
HOEXPR & ::= & hofunapp\langle\langle HOAPPLIC \times \text{seq}\ EXPR\rangle\rangle \\
& | & lambda\langle\langle VARDECL \times EXPR\rangle\rangle \\
& | & hopair\langle\langle EXPR \times EXPR\rangle\rangle \\
& | & hovar\langle\langle NAME \times SORT\rangle\rangle \\
HOAPPLIC & ::= & hofunctor\langle\langle NAME \times SORT\rangle\rangle \\
& | & expr\langle\langle EXPR\rangle\rangle
\end{array}
\end{array}
$$

$$
\begin{array}{lcl}
SET & ::= & nat \\
& | & bool \\
& | & int \\
& | & tfun\langle\langle SET \times SET\rangle\rangle \\
& | & pfun\langle\langle SET \times SET\rangle\rangle \\
& | & setcomp\langle\langle VARDECL \times FMLA\rangle\rangle \\
& | & pow\langle\langle SET\rangle\rangle \\
& | & setenum\langle\langle \text{seq}\ EXPR\rangle\rangle \\
& | & union\langle\langle SET \times SET\rangle\rangle \\
& | & intersect\langle\langle SET \times SET\rangle\rangle \\
& | & diff\langle\langle SET \times SET\rangle\rangle \\
& | & interval\langle\langle EXPR \times EXPR\rangle\rangle \\
& | & cross\langle\langle SET \times SET\rangle\rangle \\
& | & seq\langle\langle SET\rangle\rangle \\
& | & setvar\langle\langle NAME \times SORT\rangle\rangle \\
SORT & ::= & natSort \\
& | & intSort \\
& | & boolSort \\
& | & seqSort\langle\langle SORT\rangle\rangle \\
& | & setSort\langle\langle SORT\rangle\rangle \\
& | & tfunSort\langle\langle SORT \times SORT\rangle\rangle \\
& | & pfunSort\langle\langle SORT \times SORT\rangle\rangle \\
& | & pairSort\langle\langle SORT \times SORT\rangle\rangle \\
& | & sortvar\langle\langle NAME\rangle\rangle
\end{array}
$$

Figure 4: Generic Framework Syntax Description

In the IML, types are modelled using sets, with each variable assigned a set of values that it can take when it is declared. Defining types using sets means that there is no loss of information when we are translating from an FDE with a weakly-typed logic. However several predefined sets (naturals, booleans and integers) and set constructors (total functions, partial functions, power sets, cross product and sequences) are defined so that strongly-typed logics can be easily translated. A number of set operations are included for constructing specialised or more complex sets, including: set comprehension; union; intersection; and set difference. In cases where no type information can be derived, or where type parameters are required, a set variable can be used.

Sorts (*SORT*) are used for a kind of weak typing, used internally by translators to handle pathological cases and to deal with operator overloading. Sorts can be assigned to variables, functions and relations. We use sorts instead of sets for this internal processing because they are easier to calculate and provide sufficient information for internal processing. Sorts are not directly translated as part of a theorem. A number of sorts are defined, including: natural numbers; integers; booleans; sequences; sets; total functions; partial functions; and ordered pairs. Sort variables are also available for representing generic or unknown sorts.

5.2 Writing translators

Developing translators from an FDE to the IML and from the IML to a theorem prover is largely dependent on the particular FDE or theorem prover. However the framework is designed to support systematic development of translators, by: providing tables of supported constructs to make it easier to match FDE/theorem prover constructs with IML constructs; identifying standard construct adaptation techniques for matching FDE/theorem prover constructs against similar IML constructs; and using sorts to help identify and deal with pathological translation cases.

FDE to IML translators

FDE to IML translators are extended parsers that map FDE constructs into IML syntax. As well as performing standard parsing, a translator also needs to perform the following steps:

1. Extract sort information for each variable and functor.
2. Extract typing information for any variable declarations.
3. For each FDE construct find a corresponding construct in the support table, using sort information to solve any ambiguities.

4. If there is no direct corresponding construct, determine whether the FDE construct can be adapted to match an IML construct. Possible adaptations include:
   (a) modifying the index for list/sequence dereferencing,
   (b) reversing the order of arguments for binary ordering relations.
5. Rename variables to ensure there are no name clashes.

IML to theorem prover translators

An IML to theorem prover translator maps IML constructs to constructs that can be manipulated by the target theorem prover. IML to theorem prover translators are extended pretty printers, with the following steps:

1. Map each construct in IML tables to a construct in the theorem prover.
2. For unsupported constructs attempt to adapt the IML construct to match an existing theorem prover construct.
3. Translate type information for variables using the set information given in the variable declaration.
4. Use sort information to deal with any pathological cases particular to the target prover, e.g., the treatment of partial functions in Isabelle/HOL.

5.3 Tool support

Prototype tool support has been developed for the framework, written mostly in Prolog. To date the tool includes translators for the B and CARE FDEs, and the theorem provers Ergo, Isabelle/HOL and Otter (the latter being an automated prover). The prototype tool has been applied to two complete case studies in B (the stack data structure given earlier and a square root approximation program) and one complete case study in CARE (square root approximation) to generate proof obligations in Isabelle/HOL. These proof obligations were compared to those originally translated by hand to Isabelle/HOL. The proof obligations generated by the tool are equivalent to those translated by hand. The IML to Ergo translator has also been applied to these examples to generate proof obligation in Ergo, however we have not proven these proof obligations in Ergo yet.

Figure 5 contains a fragment of the IML output for the *Pop4* proof obligation in B, generated by the B to IML translator. This IML fragment corresponds to the condition:

$$stackr \in 0 \mathinner{..} max - 1 \rightarrow \mathbb{N}$$

This condition is represented by a application of the relation *mem* to the variable *stackr* and the set $0 \mathinner{..} max - 1 \rightarrow \mathbb{N}$. Lines 1-2 begin the relation application, defining the name of the construct to be applied. Lines 3-11 define the sort of the *mem* relation. Lines 12-29 define the two arguments of the relation application. Line 12 defines the first argument as the variable *stackr*. Note that *stackr* has been renamed by prepending *iml* to avoid name clashes. Lines 13-29 construct the set defined as the second argument. It uses the function *tfun* to construct a set of functions, *interval* to construct a set of natural number intervals, and *nat* to construct the set of natural numbers. The interval is constructed in lines 21-28 using the constants 0 and 1, the variable *max* (renamed) and the function *minus*.

The tool has been tested on fragments of other case studies to test whether other specific translation issues have been dealt with sufficiently. To test whether higher-order proof obligations could be successfully translated within the framework, we translated several proof obligations used to establish the correctness of a generic accumulator module in CARE (Hemer & Lindsay 2004). These proof obligations, which included higher-order parameters, were translated to Isabelle/HOL.

Another simple case study in CARE, which included domain specific theory extensions, was used to test the frameworks support for definitional extensions. In this case the framework was able to translate to suitable constructs in Isabelle/HOL for making conservative extensions.

A simple example, modelling a switch in B, was translated using the IML to Otter translator. All proof obligations were automatically discharged by Otter.

6 Discussion

In this section we discuss enhancements that could be made to the generic framework. The first enhancement is to develop a more systematic approach to developing translators. Currently translators are developed in an ad-hoc manner, with each translator having been developed from scratch. However much of the translator is more or less the same regardless of the FDE or theorem prover. We propose developing two generic translators (for FDE to IML and IML to theorem prover). These generic translators can then be instantiated by setting certain variables or rules. The idea is similar to the way in which Proof General (Aspinall 2000) enables script-based theorem prover interfaces to be easily developed by setting certain variables and functions.

Another area that we wish to investigate is the number of constructs given in the supported constructs tables. In the current version of the framework these tables are minimal. For example rather than providing two list-like structures indexed from zero and one, we only provide one. To translate from or to the other list structure we must provide an adaptation. In some cases we might have to provide an adaptation when translating from the FDE to the IML and when translating from the IML to the theorem prover, when in fact the FDE and theorem prover constructs are equivalent. This unnecessary adaptation could result in proof obligations that are harder to read and reason about. An alternative to the current approach is to include many more constructs in the tables. In translating from the FDE to the IML we would need some way of representing alternate solutions. When translating from the IML to the theorem prover we would choose the alternative that maps most naturally to a theorem prover construct.

The intermediate language used by our framework has no formal semantics. All translations via the framework are purely syntactic transformations. It may be the case that an IML with a fully formal semantics could be incorporated into the framework. This would allow us to ensure the soundness of translations. While development of an IML with a fully formal semantics may be possible (perhaps similar to the meta language of Isabelle), writing translators to and from such a formal language would be impractical, thus removing any of the advantages of using the framework.

Several translation issues are still unresolved in

```
rel(
   mem,
   tfunsort(
      pairsort(
         tfunsort(natsort,natsort),
         tfunsort(
            pairsort(
               tfunsort(pairsort(natsort,natsort),setsort(natsort)),
               setsort(natsort)),
            tfunsort(tfunsort(pairsort(natsort,natsort),setsort(natsort)),setsort(natsort)))),
      boolsort),
   [var(iml_stackr,tfunsort(natsort,natsort)),
   funapp(
      tfun,
      tfunsort(
         pairsort(
            tfunsort(pairsort(natsort,natsort),setsort(natsort)),
            setsort(natsort)),
         tfunsort(tfunsort(pairsort(natsort,natsort),setsort(natsort)),setsort(natsort))),
      [funapp(
          interval,
          tfunsort(pairsort(natsort,natsort),setsort(natsort)),
          [funapp(0,natsort,[]),
           funapp(
              minus,
              tfunsort(pairsort(natsort,natsort),natsort),
              [var(iml_max,natsort),
               funapp(1,natsort,[])])]),
       funapp(nat,setsort(natsort),[])]))
```

Figure 5: IML representation for Pop4 proof obligation fragment

the current version of the framework. One of these is the area of non-definitional theory extensions, i.e., constructs that have an associated proof. For this work we would need a generic proof representation (Watson 2001), but it is not clear that this would be possible without formalising the semantics of the IML. Another issue that is still open is the treatment of undefinedness. While we are able to translate to theorem provers that provide explicit support for modelling undefinedness (e.g., Ergo), we cannot represent proof obligations that make explicit use of the undefined construct. Again it is not clear that such an extension could be made to the framework without formalising the semantics of the IML, in this case by introducing a three-valued logic.

7 Conclusions

In this paper we described a generic framework for connecting multiple formal development environments to multiple theorem provers. The requirements for this framework were influenced by three case studies that revealed a number of translation issues. These three case studies also provided concrete evidence that standalone prover support was better than the prover support provided as part of the FDE. The collection of translation issues, as described in this paper, was expanded further by completing a literature survey.

The generic framework, in which proof obligations are translated via an intermediate modelling language, is currently supported by translators for two FDEs (CARE and B) and three theorem provers (Isabelle/HOL, Ergo, and Otter). This tool support provides improved prover support for B. Moreover it is a valuable addition to the current version of the CARE toolset, which previously did not provide any prover support.

References

Abrial, J.-R. (1996), *The B Book: Assigning Programs to Meanings*, Cambridge University Press.

Agerholm, S. (1996), Translating specifications in VDM-SL to PVS, *in* J. v. Wright, J. Grundy & J. Harrison, eds, 'Theorem Proving in Higher Order Logics, 9th International Conference', Vol. 1125 of *LNCS*, Springer-Verlag, pp. 1–16.

Ahrendt, W., Baar, T., Beckert, B., Giese, M., Hahnle, R., Menzel, W., Mostowski, W. & Schmitt, P. H. (2002), The KeY system: Integrating object-oriented design and formal methods, *in* R.-D. Kutsche & H. Weber, eds, 'FASE 2002', Vol. 2306 of *LNCS*, Springer-Verlag, pp. 327–330.

Aspinall, D. (2000), Proof general: A generic tool for proof development, *in* S. Graf & M. Schwartzbach, eds, 'Tools and Algorithms for the Construction and Analysis of Systems', Vol. 1785 of *LNCS*, Springer-Verlag, pp. 38–42.

Bodeveix, J.-P. & Filali, M. (2002), Type synthesis in B and the translation of B to PVS, *in* D. Bert, J. Bowen, M. Henson & K. Robinson, eds, 'ZB 2002: Formal specification and development in Z and B. 2nd International Conference of B and Z users', Vol. 2272 of *LNCS*, Springer-Verlag, pp. 350–369.

Boulton, R., Gordon, A., Gordon, M., Herbert, J. & Tassel, J. v. (1992), Experience with embedding hardware description languages in HOL, *in* V. Stavridou, T. Melham & R. Boute, eds, 'International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience', North-Holland, pp. 129–156.

Chartier, P. (1998), Formalisation of B in Isabelle/HOL, *in* D. Bert, ed., 'Second Inter-

national B Conference', Vol. 1393 of *LNCS*, Springer-Verlag, pp. 66–83.

Clutterbuck, D., Bicarregui, J. & Matthews, B. (1996), Experiences with proof in formal development, *in* 'First International Conference on B', Institut de Recherche en Informatique de Nantes.

Craigen, D., Kromodimoeljo, S., Meisels, I., Pase, B. & Saaltink, M. (1991), Eves : an overview, *in* S. Prehn & W. Toetenel, eds, 'VDM'91', Springer-Verlag.

Dennis, L. A., Collins, G., Norrish, M., Boulton, R., Slind, K., Robinson, G., Gordon, M. & Melham, T. (2000), The PROSPER toolkit, *in* S. Graf & M. Schwartzbach, eds, '6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems', Vol. 1785 of *LNCS*, Springer-Verlag, pp. 78–92.

Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice Hall.

Elmstrom, R., Larsen, P. & Lassen, P. (1994), 'The IFAD VDM-SL Toolbox: A practical approach to formal specifications', *ACM Sigplan Notices* **29**(9), 77–80.

Finin, T., Fritzson, R., McKay, D. & McEntire, R. (1994), KQML as an agent communication language, *in* 'Proceedings of the Third International Conference on Information and Knowledge Management', ACM Press, pp. 456–463.

Franke, A., Hess, S., Jung, C., Kohlhase, M. & Sorge, V. (1999), 'Agent-oriented integration of distributed mathematical services', *Journal of Universal Computer Science* **5**(3), 156–187.

Franke, A. & Kohlhase, M. (1999), System description: Mathweb, an agentbased communication layer for distributed automated theorem proving, *in* 'Proceedings of CADE'99', Vol. 1632 of *LNCS*, Springer Verlag, pp. 217–221.

Giunchiglia, F., Bertoli, P. & Coglio, A. (1998), The OMRS project: State of the art, *in* 'Proc. 2nd Workshop on Rewriting Logic and Its Applications (WRLA'98)', Vol. 15 of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers.

Hemer, D. & Lindsay, P. (2004), 'Template-based construction of verified software', *IEE Proceedings Software* . Accepted for publication.

Jones, C. B. (1990), *Systematic Software Development using VDM*, Prentice-Hall.

Kalman, J. A. (2001), *Automated reasoning in OTTER*, Rinton Press.

Lindsay, P. & Hemer, D. (1996), An industrial-strength method for the construction of formally verified software, *in* P. A. Bailes, ed., 'ASWEC'96', IEEE Computer Society Press, pp. 27–36.

Naumov, P., Stehr, M.-O. & Meseguer, J. (2001), The HOL/NuPRL proof translator : A practical approach to formal interoperability, *in* R. J. Boulton, ed., 'Theorem Proving in Higher Order Logics, 14th International Conference', Vol. 2152 of *LNCS*, Springer-Verlag, pp. 329–345.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002), *Isabelle/HOL A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *LNCS*, Springer-Verlag.

Owre, S., Rushby, J. M. & Shankar, N. (1992), PVS: A prototype verification system, *in* D. Kapur, ed., '11th International Conference on Automated Deduction', Vol. 607 of *LNAI*, Springer-Verlag, pp. 748–752.

Paulson, L. C. (1989), 'The foundation of a generic theorem prover', *Journal of Automated Reasoning* **5**(3), 363–396.

Saaltink, M., Craigen, D., Kromodimoeljo, S. & Pase, B. (1992), Sharing is difficult, *in* 'TTCP-XTP-1 Workshop on effective use of automated reasoning technology in system development', pp. 42–47.

Schneider, S. (2001), *The B-Method: An Introduction*, Cornerstones of computing, Palgrave.

Smith, D. R. (1990), 'KIDS: A semi-automatic program development system', *IEEE Transactions on Software Engineering* **16**(9), 1024–1043.

Spivey, M. (1992), *The Z Notation: A Reference Manual*, International Series in Computer Science, 2nd edn, Prentice Hall.

Stringer-Calvert, D. W. J., Stepney, S. & Wand, I. (1997), Using PVS to prove a Z refinement: A case study, *in* J. Fitzgerald, C. B. Jones & P. Lucas, eds, 'FME'97', Vol. 1313 of *LNCS*, Springer-Verlag, pp. 573–588.

Utting, M., Robinson, P. & Nickson, R. (2002), 'Ergo6: a generic proof engine that uses Prolog proof technology', *LMS Journal of Computation and Mathematics* **5**, 194–219.

Watson, G. N. (2001), A Generic Proof Checker, PhD Thesis, University of Queensland.