To be presented at FM'99: The World Congress on Formal Methods
Toulouse, France, September 1999.

Structural Embeddings: Mechanization with Method

César Muñoz[1] and John Rushby[2] *

[1] Institute for Computer Applications in Science and Engineering (ICASE)
Mail Stop 132C, 3 West Reid Street
NASA Langley Research Center
Hampton VA 23681-2199
munoz@icase.edu

[2] Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

Abstract. The most powerful tools for analysis of formal specifications are general-purpose theorem provers and model checkers, but these tools provide scant methodological support. Conversely, those approaches that do provide a well-developed method generally have less powerful automation. It is natural, therefore, to try to combine the better-developed methods with the more powerful general-purpose tools. An obstacle is that the methods and the tools often employ very different logics.

We argue that methods are separable from their logics and are largely concerned with the structure and organization of specifications. We propose a technique called *structural embedding* that allows the structural elements of a method to be supported by a general-purpose tool, while substituting the logic of the tool for that of the method. We have found this technique quite effective and we provide some examples of its application. We also suggest how general-purpose systems could be restructured to support this activity better.

1 Introduction

In recent years, the capabilities of theorem provers oriented towards support of formal methods (we call them *verification systems*) have increased enormously. Systems such as ACL2 [23], Coq [5], Eves [42], HOL [27], Isabelle [36], and PVS [30] each come with a very rich specification language and a battery of decision procedures and proof strategies highly tuned to their logic. Some also provide convenient access to model checkers or to specialized decision procedures through built-in embeddings and interpretations, and some are able to generate

* This work was supported by National Science Foundation grant CCR-9509931 while the first author was an International Fellow at SRI International.

efficiently executable code. This integration of rich specification languages with powerful automation allows general-purpose verification systems to attack very complex problems in a broad spectrum of domains [40].

A commonly-cited drawback to the use of these systems, is their lack of methodological support for the global process of specification and software development: with their emphasis on deductive support, the overall structure of a development is relegated to an external (informal or formal) methodology with little automated support. For this reason, some people complain that there is little method in formal methods.

On the other hand, formal notations such as B [1], VDM [22], Z [44], and the requirements methodologies that employ tabular specifications [19,25,43] emphasize the methodological aspects of software specification and development. That is to say, they suggest how specifications should be structured and organized, how different specifications should be related to each other and to executable programs, and what theorems (i.e., "proof obligations") should be posed and proved in order to gain confidence in a specification or in the correctness of a refinement. These methods provide a formal notation and sometimes provide automated support for their methodological aspects, but usually their logic is supported only by relatively limited and specialized theorem provers, so that it can be tedious to discharge proof obligations, and difficult to establish properties of the overall specification.

It is natural to ask whether the complementary strengths of general-purpose verification systems and of the more methodical formal notations can be combined in some way. One way to do this is by a *semantic embedding* of the formal notation within the logic of the verification system. Two variants have been identified: *deep* and *shallow* embeddings [10].

In a deep embedding, the language and semantics of the method are fully formalized as an object in the logic of the specification language. In this case, it is possible to prove meta-theoretical properties of the embedded method, but the statement and proof of properties for a particular application require painful encoding into the formalized semantics. In the shallow approach, there is a syntactic translation of the objects of the method into semantically equivalent objects in the language of the verification system. In this case, meta-theoretical properties cannot be stated, but the encoding and analysis of particular applications is simpler.

Both of these approaches consider the formal notation as a unity and do not separate method from logic. This is consistent with the way most formal methods are presented—the methodological aspects of B, for example, are described in terms of a certain set theory [1], and a certain logic of partial terms is introduced to support the method of tabular specifications [34].

We question whether such unity—the tight coupling of method and logic—really is necessary. To our thinking, the method-specific aspects tend to be at the outermost, or "structural" levels of the specification language, and are not very sensitive to the actual logic employed for expressions inside the structure.

For example, the tabular method employs tables to specify aspects of a system's requirements or behavior, but is largely indifferent to the logic in which table entries are specified, provided that it possesses certain attributes (e.g., an adequate treatment of partial functions).

Given this perspective, we propose a new kind of embedding, in which the structural part of a method is embedded in the logic of the verification system (by means of either a shallow or a deep embedding, but most commonly the former), while the logic part of the method (its notation for expressions) is simply *replaced* by that of the verification system. By fitting the structural language elements of a method around a well-supported logic, we get the best of both worlds, and quite cheaply. Of course, this will not satisfy those who require the authentic language of a particular formal method, but it provides an attractive way to support the "style" of such a method, or to add methodological discipline to the raw logic of a verification system.

In this paper we study this variation on embedding, which we call *structural embedding*. The paper is organized as follows. We give an overview of the notions involved in this kind of embedding in Section 2 and we describe examples in Sections 3, and 4. The final section compares this approach with others, and discusses how general-purpose verification systems could be restructured to better support this type of activity.

2 Structural Embedding

A formal method provides a specification language, which is built on a particular logic. Since formal methods are intended to organize formal specifications, the specification language is invariably structured in several syntactic levels. Usually, the outermost level concerns some notion of "module" and relationships among these, while the innermost level provides the expression language.

Different names are used for the top-level module constructs in different specification languages: for example, machines in B, schemas in Z, theories in PVS. Specification languages usually provide several mechanisms to combine their modules in order to build large-scale systems. Most of the method in a formal method is expressed at this level. For example, invariants may be specified at the module level, giving rise to proof obligations on the operations specified within each module, or refinement relationships may be specified across modules, giving rise to further proof obligations.

An *embedding* is a semantic encoding of one specification language into another, intended to allow tools for the one to be extended to the other. In our context, we are interested in embedding the specification language of a formal method into that of a verification system. Using embeddings, the complementary strengths of several formal methods and verification systems can be combined to support different aspects of verified software development.

The semantics of the language of a formal method can be encoded in a verification system either by using an extra-logical translation (i.e., a kind of compiler), in which case we speak of a *shallow* embedding; or it can be defined directly in the specification language of the verification system, and in this case we talk of a *deep* embedding [10]. In a *structural* embedding, which is orthogonal to both of these, only the outermost level of the specification language is embedded in the logic of the verification system. The innermost level of the specification language is directly *replaced*, not embedded, by the expression language of the verification system. The logical framework of the embedded notation relies completely on the specification language of the verification system.

We can describe the way this works as follows. Let $\mathcal{L}_{\text{FM}}$ and $\mathcal{L}_{\text{VS}}$ be the specification languages of a formal method and a verification system, respectively. By language abuse, we use the same symbols for their logics. We use the judgment $S \models_{\mathcal{L}} P$ to mean that P is a property satisfied by the specification S in the logic $\mathcal{L}$. In these terms, a semantic embedding is a translation $\_^* : \mathcal{L}_{\text{FM}} \mapsto \mathcal{L}_{\text{VS}}$ satisfying

$$S \models_{\mathcal{L}_{\text{FM}}} P \;\;\Rightarrow\;\; \mathcal{L}_{\text{FM}}\_in\_\mathcal{L}_{\text{VS}} \wedge S^* \models_{\mathcal{L}_{\text{VS}}} P^*$$

where $\mathcal{L}_{\text{FM}}\_in\_\mathcal{L}_{\text{VS}}$ is the set of axioms and definitions in $\mathcal{L}_{\text{VS}}$ encoding the semantics of $\mathcal{L}_{\text{FM}}$. The shallow or deep degree of the embedding depends on the information contained in $\mathcal{L}_{\text{FM}}\_in\_\mathcal{L}_{\text{VS}}$.

For a structural embedding, we consider that $\mathcal{L}_{\text{FM}}$ consist of two sub-languages $\mathcal{L}_{\text{FM}} = \mathcal{L}^o_{\text{FM}} \cup \mathcal{L}^i_{\text{FM}}$, where $\mathcal{L}^o_{\text{FM}}$ represents the outermost level of language, and $\mathcal{L}^i_{\text{FM}}$ represents the innermost one. First, we construct $\mathcal{L}'_{\text{FM}} = \mathcal{L}^{o\,\prime}_{\text{FM}} \cup \mathcal{L}_{\text{VS}}$, which replaces the inner language by that of the verification system and adjusts $\mathcal{L}^o_{\text{FM}}$ (as $\mathcal{L}^{o\,\prime}_{\text{FM}}$) to accommodate its new context while preserving its "intent." There is no formal relationship or mechanical translation between $\mathcal{L}_{\text{FM}}$ and $\mathcal{L}'_{\text{FM}}$—the goal is simply to preserve the ideas and intent of the method to the extent possible.

A structural embedding is then a translation $\_^* : \mathcal{L}^{o\,\prime}_{\text{FM}} \mapsto \mathcal{L}_{\text{VS}}$, which is extended to $\_^* : \mathcal{L}'_{\text{FM}} \mapsto \mathcal{L}_{\text{VS}}$ in the obvious way (as the identity on $\mathcal{L}_{\text{VS}}$) satisfying

$$S \models_{\mathcal{L}'_{\text{FM}}} P \;\;\Rightarrow\;\; \mathcal{L}^{o\,\prime}_{\text{FM}}\_in\_\mathcal{L}_{\text{VS}} \wedge S^* \models_{\mathcal{L}_{\text{VS}}} P^*$$

where $\mathcal{L}^{o\,\prime}_{\text{FM}}\_in\_\mathcal{L}_{\text{VS}}$ is the set of axioms and definitions in $\mathcal{L}_{\text{VS}}$ encoding the semantics of $\mathcal{L}^{o\,\prime}_{\text{FM}}$. Notice that the semantics of $\mathcal{L}^i_{\text{FM}}$ are not embedded, and that both of shallow and deep embedding are still possible for $\mathcal{L}^{o\,\prime}_{\text{FM}}$.

To preserve intent in a structural embedding requires that well-formedness of specifications is preserved in both logics. That is,

$$\models_{\mathcal{L}_{\text{FM}}} \mathit{Sound}_{\mathcal{L}_{\text{FM}}}(S) \;\;\Leftrightarrow\;\; \mathcal{L}'_{\text{FM}}\_in\_\mathcal{L}_{\text{VS}} \models_{\mathcal{L}_{\text{VS}}} \mathit{Sound}_{\mathcal{L}_{\text{VS}}}(S^*).$$

By $\mathit{Sound}_{\mathcal{L}}(S)$, we mean the set of formulas (proof obligations) that guarantees some method-specific well-formedness property of specification S in logic $\mathcal{L}$ (e.g., the checks for overlapping or missing conditions in a tabular specification). Formal methods are often concerned with metalogical relationships between specifications (e.g., that one should be a refinement of another, or that one should be an invariant for the other), and *Sound* is then extended to the proof obligations that ensure satisfaction of the desired relationship. Notice that *Sound* is parameterized by the logic. In practice, we expect that *Sound* relies only on very general properties of a logic, so that proof obligations retain their intuitive content under the structural embedding.

In the following two sections we present concrete examples of structural embeddings.

3 The B-Method in PVS

In this first example, we describe a structural embedding of the B-method in the higher-order logic of PVS.

The B-method [1] is a state-oriented formal method mainly intended for development of sequential systems. The underlying logic of the method is a set theory with a first-order predicate calculus. PVS [30] is a verification system whose specification language is a higher-order logic with a type system. PVS does not come with a particular built-in methodology.

3.1 An Overview of the B-Method

In B, specifications are structured in modules called *machines*. Machines can be of three kinds: *abstract machines*, *refinements*, and *implementations*. Each kind of machine corresponds to a different stage of software development. The initial specification of a problem is given by a set of abstract machines. Refinements allows data reification of specifications. Final refinements, those that are not intended to be refined anymore, are called implementations.

A machine is an abstract description of the statics and dynamics of a system. Statics are given by a state declaration: constants, properties of the constants, variables, and an invariant (a property satisfied by the state of the machine). Dynamics are given by operations or services provided by the machine. In contrast to other stated oriented methods, operations in B are not specified by *before-after* predicates, but by an equivalent mechanism of predicate transformers called *generalized substitutions*.

Large software development is supported using several composition mechanisms. These mechanisms give different access privileges to the operations or to the local variables of an external machine. In this way, it is possible to build complex machines incrementally by using previously defined ones. Thus, by using the unified notation of machines, B supports the complete life cycle of software development.

Several cases studies of developments in B are reported in [7]. That work pointed out some drawbacks of the B-method:

- Although typing conditions can be handled using the set theory provided by B, mathematical objects such as variables or functions are not explicitly typed. In some cases this “free-typing” style obscures the specifications.

– The generalized substitutions mechanism encourages the writing of algorithmic specifications. Some kind of operations could be more naturally expressed by before-after predicates. The same conclusion was drawn by Bicarregui and Ritchie in [8].
– Support for data types is limited. In particular, record types are absent in the B notation.
– Proof obligations usually deal with type conditions that could be easily solved by a type checker.
– B imposes a very rigid discipline. For instance, parameters of a machine are restricted to be scalars or uninterpreted sets. In some cases such restrictions seem to be very strong.

Most of these criticisms concern the limitation of the formal notation rather than the methodological aspects of B. We argue that it is possible to separate the abstract machine mechanism from its specification language, and to use the expression language of PVS instead of that of B. In this way, we combine the best features of each technique: the methodology of B, and the expressiveness and richness (and automation) of the specification language of PVS.

3.2 An Example: a Drinks Dispenser Machine

To concretize our ideas, we present in Figure 1 an example of a drinks dispenser specification written in B by Leno and Haughton [24]. The specification is, for most of the parts, self-explanatory.

At first glance, the expressions of the machine `Dispenser` could be easily translated to PVS. For instance, the invariant

$$\mathtt{dstate} \in \mathtt{DSTATE} \wedge \mathtt{given} \in \mathtt{NAT} \wedge \mathtt{given} \leq \mathtt{lifetime}$$

literally corresponds to the PVS expression

```
member(dstate,DSTATE) AND member(given,NAT) AND given <= lifetime.
```

However, the PVS specification language is fully-typed while the B notation is not. For instance, although it is possible to define a set in PVS containing all the natural numbers, the normal way to handle a property like `given ∈ NAT` in PVS is by using a type declaration `given:NAT`—the natural numbers are a basic type in PVS, whereas they are a predefined set in B.[1] Thus, in PVS, the invariant is reduced to

```
given <= lifetime.
```

and its other two clauses become typing judgments.

[1] In fact, in B, `NAT` is the predefined set of naturals numbers between 1 and `maxint`, where `maxinit` is not known a priori. PVS can also represent this as a type: `subrange(1,maxint)`.

```
MACHINE Dispenser(lifetime)
  SETS
    DSTATE = { stocked, unstocked }

  CONSTANTS
    ok, notok

  PROPERTIES
    ok = 0 ∧ notok = 1

  VARIABLES
    dstate, given

  INVARIANT
    dstate ∈ DSTATE ∧ given ∈ NAT ∧ given ≤ lifetime

  INITIALIZATION
    dstate := unstocked ||
    given := 0

  OPERATIONS
     restock =
       dstate := stocked;

     give_drink =
       PRE dstate = stocked ∧ given < lifetime THEN
           dstate :∈ DSTATE ||
           given := given+1
       END;

     bb ⟵ is_stocked =
       IF dstate = stocked THEN
          bb := ok
       ELSE
          bb := notok
       END;

     count ⟵ number_given =
       count := given

END
```

Fig. 1. A Drinks Dispenser in B

In Figure 2 we present a fully typed version of the dispenser machine which uses the expression language of PVS. Notice also that PVS machines use a clause `TYPES` rather than the original clause `SETS` of B. From the PVS point of view, `DSTATE` is not a set, but a type. Its role in the specification is not that of a container, but that of a typing tag. Also note that functions are not interpreted as binary relations in PVS, but as computational objects.

3.3 Semantics

The semantics of the B-method is described in [1] in terms of a particular set theory and a first-order logic. Roughly speaking the soundness of a specification is given by the validity of a set of axioms extracted from the machines. These axioms are usually called *proof obligations*. The more important axioms concern the preservation of the invariant by the operations. In general, these proof obligations have the form:

$$\texttt{PROPERTIES} \wedge \texttt{INVARIANT} \Rightarrow \texttt{[OPERATION]INVARIANT.}$$

As noted before, operations are defined in B as predicate transformers. Thus, for example, the proof obligation concerning the initialization clause of the machine `Dispenser` states that after the initialization of the machine, the invariant is satisfied. Formally, it states that the following proposition holds:

$$\texttt{ok = 0} \wedge \texttt{notok = 1} \Rightarrow \texttt{[dstate:=unstocked} \parallel \texttt{given:=0]INVARIANT.}$$

That is

$$\texttt{ok = 0} \wedge \texttt{notok = 1} \Rightarrow \texttt{unstocked} \in \texttt{DSTATE} \wedge \texttt{0} \leq \texttt{lifetime,}$$

which is trivially true.[2]

As pointed out before, a major difference between the specifications given in Figures 1 and 2 is that PVS machines are based on the higher-order logic and type theory of PVS. In particular, a B machine is embedded as a PVS theory, where the parameters and types of the machine become parameters and types of the theory.

The state of a B machine is encoded in the functional style of PVS as follows. The variables of the machine define a record type, called the *general type*. Each field of the record corresponds to a variable of the machine. The invariant of the machine is expressed as a subtype of the general type. In this way, the mutual dependence between the variables given by the constraints is handled by the dependent type mechanism of PVS.

[2] In B, lowercase parameters, as `lifetime`, are assumed to be scalars.

```
Dispenser_in_PVS [ lifetime:nat ] : MACHINE
BEGIN
  TYPES
    DSTATE = {stocked, unstocked}

  CONSTANTS
    ok : nat = 0
    notok : nat = 1

  VARIABLES
    dstate : DSTATE
    given : nat

  INVARIANT
    given <= lifetime

  INITIALIZATION
    dstate := unstocked ||
    given := 0

  OPERATIONS
    restock =
      dstate := stocked

    give_drink =
      PRE dstate = stocked AND given < lifetime THEN
         dstate :: DSTATE ||
         given := given + 1
      END

    is_stocked : nat =
      IF dstate = stocked THEN
         ok
      ELSE
         notok
      ENDIF

    count : nat =
      given

END Dispenser_in_PVS
```

Fig. 2. The Drinks Dispenser Machine Structurally Embedded in PVS

The general type defined for `Dispenser_in_PVS` is

```
Dispenser_in_PVS_Type : TYPE = [#
  dstate:DSTATE,
  given:nat
#]
```

(Record types in PVS are declared by using the brackets `[#,#]`. Instances of a record type are given between `(#,#)` parentheses. Record and function overriding are indicated in PVS by the `WITH` construct.)

The invariant of the machine is handled by the following type:

```
Dispenser_in_PVS : TYPE =
   { self: Dispenser_in_PVS_Type | given(self) <= lifetime }
```

An operation `op` of a machine M with inputs $\mathtt{i}_1\mathtt{:I}_1\mathtt{,...,i}_n\mathtt{:I}_n$ and outputs $\mathtt{o}_1\mathtt{:O}_1\mathtt{,...,o}_m\mathtt{:O}_m$, is translated into PVS as a function $\mathtt{op(i}_1\mathtt{:I}_1\mathtt{,...,i}_n\mathtt{:I}_n\mathtt{)(self:M):[o}_1\mathtt{:O}_1\mathtt{,...,o}_m\mathtt{:O}_m\mathtt{,self\_out:M]}$. If `op` has no inputs and outputs, its signature is simply `op(self:M):M`. For instance:

```
restock(self:Dispenser_in_PVS) : Dispenser_in_PVS =
  LET self =
    self WITH [
      dstate := stocked
    ] IN
  self
```

Generalized substitutions are interpreted as PVS expressions dealing with record field overriding, function updating, set operations, and typing conditions. Certain kinds of compositions are supported by using the importing mechanism of PVS. The complete embedding is described in [28].

Soundness of a B machine corresponds to type correctness of the PVS theory embedding it. Therefore, the proof obligations to be checked are just the type correctness conditions (TCCs) generated by the PVS type system, and so it is possible to use the automation provided by the PVS type-checker and theorem prover. The type correctness conditions generated for the PVS embedding of a B machine guarantee that the initial state satisfies the invariant and that the invariant is preserved by the operations.

PVS generates four TCCs for the machine `Dispenser_in_PVS`. All of them are automatically discharged by the theorem prover. For instance, the TCC corresponding to the initialization clause is

```
init_TCC1 :

  |-------
{1} (∀ (self):
          self = (# dstate := unstocked, given := 0 #) ⇒
          0 ≤ lifetime)
```

The embedding that we have described corresponds to a shallow structural embedding. That is, meta-theoretical properties about the abstract machine notation cannot be proved. It has been completely implemented by a front-end tool called PBS [28]. An alternative deeper embedding has been proposed in [9]. That work formalizes the generalized substitution mechanism of the B-method in the higher-order logic of Coq and PVS. In this case, it is possible to verify meta-theoretical properties about generalized substitutions.

3.4 The PBS System

PBS works like a compiler. It takes as input a file *m.bps* containing an abstract machine and generates its corresponding embedding as a PVS theory in the file *m.pvs*. We have rewritten several examples of abstract machines from [1, 24, 29] in PBS. The results obtained are satisfactory according to our expectations: trivial type conditions are discharged automatically by the type checker of PVS, and most of the other proof obligations can be solved by the automated decision procedures and strategies provided by its theorem prover.

Table 1 summarizes one of these developments. `Client`, `Product`, and `Invoice` are part of an invoice system developed in [1]. The example provides the basic functionality of a data processing system. During the development, the type checker of PVS allowed us to find some minor errors in the specification given in [1].

Machine	PBS (in lines)	PVS theory (in lines)	TCCs	Auto proved
`Client`	56	83	12	100%
`Product`	66	92	18	83%
`Invoice`	125	166	48	87%

Table 1. Metrics of Some Examples

Büchi [11, 12] describes a prototypical banking application implemented in two commercial tools supporting the B-method: Atelier B from Steria and the B-Toolkit from B-Core. `Bank` is the largest machine of that system, and we have rewritten it in PBS. In Table 2, we compare our metrics for this example with those given by Büchi. [3]

The difference between the size of the files is due to the fact that many properties are attached to the types of the variables and parameters in the PBS specification and therefore need not be repeated in the invariant and the pre-conditions to the operations, making the specification shorter. The proof obligations of the PBS and B machines do not correspond one-to-one either: recall that proof obligations in PBS machines are generated by the type checker

[3] For these developments we are using PVS Version 2.3.

Machine	File length (in lines)	Proof obligations	Auto proved
Bank in PBS	232	47	94%
Bank in B	362	49	95%

Table 2. Comparison Between B and PBS Machines

of PVS, which is able to solve some type conditions internally, and to subsume some type conditions in others.

A feature introduced in PVS Version 2.3 allows PVS "ground terms" (i.e., executable definitions applied to concrete data) to be evaluated via compilation into Lisp. The compiler (due to N. Shankar) uses sophisticated static analysis to eliminate some of the inefficiencies of applicative programs, so that compiled PVS executes extremely rapidly. Combined with the refinement mechanism of the B-Method, this provides good support for rapid prototyping, testing, and code generation. For example, by refining the PVS choice function that interprets the `ANY` construct of B into a linear search, we obtain a rapid prototype for the B-Bank that can perform many thousand Bank operations (create an account, make a deposit, perform a balance enquiry, etc.) per second.

PBS and some of the examples that we have developed are available electronically at: `http://www.csl.sri.com/~munoz/src/PBS`.

4 Tabular Representations

Several methods for documentation and analysis of requirements make some use of tabular specifications. These include methods such as SCR and CoRE that are derived from the "four variable model" of Parnas [35], the RSML notation of Leveson [25], and the decision tables of Sherry [43]. All these methods can be considered as having two levels of "structure" above their base logic: the top level provides the attributes that are unique to each method, but the lower level is broadly similar across all of them: it is the use of tables to define functions by cases. A simple example is the following definition of the function $sign(x)$, which returns $-1, 0$, or 1 according to whether its integer argument is negative, zero, or positive.

$$sign(x) = \begin{array}{|c|c|c|}\hline x < 0 & x = 0 & x > 0 \\ \hline -1 & 0 & +1 \\ \hline \end{array}$$

This is an example of a piecewise continuous function that requires definition by cases, and the tabular presentation provides two benefits.

- It provides a visually attractive presentation of the definition that eases comprehension.
- It makes the cases explicit, thereby allowing checks that none of them overlap and that none have been forgotten.
  The checks for forgotten and overlapping cases generate proof obligations that have been shown to be a potent tool for error detection [19].

The structural properties of tables interact with well-definedness concerns for the underlying logic, as seen in the following table from [33, Figure 1] where the applications of the (real-valued) square root function in the second and third rows can only be shown to be well-defined (that is, to have nonnegative arguments) when the corresponding row constraints are taken into account.

	$y = 27$	$y > 27$	$y < 27$
$x = 3$	$27 + \sqrt{27}$	$54 + \sqrt{27}$	$y^2 + 3$
$x < 3$	$27 + \sqrt{-(x - 3)}$	$y + \sqrt{-(x - 3)}$	$y^2 + (x - 3)^2$
$x > 3$	$27 + \sqrt{x - 3}$	$2 \times y + \sqrt{x - 3}$	$y^2 + (3 - x)^2$

Another interaction is seen when tables allow "don't care" and blank entries (which must be shown to be unreachable).

An example of the latter is the quotient lookup table for an SRT divider shown at right. The notorious Pentium `FDIV` bug was due to bad entries in similar table. The triangular-shaped blank regions at top and bottom of these tables are never referenced by the division algorithm; the Pentium error was that certain entries believed to be in this inaccessible region, and containing arbitrary data, were, in fact, sometimes referenced during execution [37]. Proof obligations to show that such regions truly are unreachable can help avoid such errors [26, 39].

Notice that the logic required to provide an interpretation for tables with blank entries must be one that provides either partial functions, or dependent typing.

	000	001	010	011	100	101	110	111
01010								2
01001						2	2	2
01000					2	2	2	2
00111			2	2	2	2	2	2
00110		2	2	2	2	2	2	2
00101	2	2	2	2	2	2	2	1
00100	2	2	2	2	c	1	1	1
00011	2	c	1	1	1	1	1	1
00010	1	1	1	1	1	1	1	1
00001	1	1	1	1	e	0	0	0
00000	0	0	0	0	0	0	0	0
11111	0	0	0	0	0	0	0	0
11110	−1	−1	d	d	0	0	0	0
11101	−1	−1	−1	−1	−1	−1	−1	−1
11100	a	b	−1	−1	−1	−1	−1	−1
11011	−2	−2	−2	b	−1	−1	−1	−1
11010	−2	−2	−2	−2	−2	−2	b	−1
11001	−2	−2	−2	−2	−2	−2	−2	−2
11000			−2	−2	−2	−2	−2	−2
10111				−2	−2	−2	−2	−2
10110						−2	−2	−2
10101							−2	−2

Parnas [34] proposes a partial term logic similar to that of Beeson [6, Section 5] for dealing with these complexities. Parnas' approach is perfectly satisfactory, but we contend that tables are a structural element that can be hosted, with suitable adjustments and restrictions, on almost any logic.

In particular, the predicate and dependent typing of PVS [41], although quite different to Parnas' logic, provides an adequate foundation for a very rich set of tabular constructions. The structural embedding of tables into PVS is a shallow one that differs from the PBS embedding of B by being integrated directly into PVS using an intermediate `COND` construct [31]. It would have been perfectly feasible to use an external translation similar to that of PBS, but tables seemed of sufficiently general utility that we preferred a more tightly integrated implementation. The specific tabular constructions of SCR, RSML, and Sherry can then be encoded into the generic PVS tables using techniques described in [31].

The structural embedding of tables in PVS can be compared with an alternative approach where theorem provers have been used as back-ends to method-specific table analyzers. One example is RSML, where proof obligations generated by a dedicated tool have been submitted to a BDD-based tautology checker [18], PVS [17], and the Stanford Validity Checker (SVC) [32]. In all these cases, the back-end tools are used only to examine proof obligations that ensure no overlapping or forgotten cases: they do not have access to other specification properties (e.g., they would not be able to state or prove that $sign(x)$ is idempotent). With the structural embedding in PVS, however, the full specification is available for analysis; [31] describe examples where PVS is used to analyze (by theorem proving and model checking) properties of tabular specifications that extend beyond simple consistency of the tables themselves.

5 Comparison, Recommendation, and Conclusion

A formal *method* provides guidance and discipline in the application of formal mathematics to the processes of specification, design, and implementation of software and hardware systems. Verification systems, theorem provers, and model checkers can provide mechanized support for the analysis of such formal descriptions. If we want both method and mechanization, there seem to be four basic choices.

- Develop mechanized support for the chosen method from the ground up. The B tools exemplify this approach.
- Develop front-end tools for the chosen method and use existing verification systems and model checkers for back-end reasoning support. For example, the front end tools may generate proof obligations that are submitted to a theorem prover. Some of the tools developed for RSML and SCR exemplify this approach.
- Provide an embedding of the chosen method into the logic supported by a verification system. Embeddings of VDM in PVS and Isabelle exemplify this approach.
- Add method to an existing verification system or model checker. Structural embeddings are one way to do this: we take the structural or "method" level of the language from an existing method and wrap it around the logic of a verification system (or, dually, we take an existing method and replace the "logic" level of its language by that of a verification system). The structural embedding of B in PVS by the PBS tool exemplifies this approach.

The "ground up" approach potentially can deliver the most seamless integration, but incurs the very high cost of developing a customized theorem prover for the chosen method. It is not just that theorem provers are large and complex tools, and therefore expensive to develop and maintain. The largest cost is the hidden one of gaining the experience necessary to build an effective theorem prover: these systems require delicate judgments concerning how to integrate interaction and automation, how to combine rewriting and decision procedures, how to decide combinations of theories, how to integrate decision procedures with heuristics, and how to combine an expressive notation with effective deductive support. It is no accident that the most effective verification systems come from groups that have been building them for a decade or more, and that have learned from many failures.

The "back-end" approach can be an effective way to discharge proof obligations, but does not allow the verification system to provide any other kind of deductive support. For example, as noted, the RSML table analyzer generates proof obligations that have been submitted to several different theorem proving components, but these tools see only the proof obligations and do not have access to the full specification. When a different kind of analysis is desired—for example, checking of invariants—then a different translator and a different back end tool (e.g., a model checker) may be required [13]. By contrast, the structural embedding of tables in PVS allows all the capabilities of PVS to be applied to the full specification, including use of its model checker to examine invariants [31].

Checking of proof obligations with a back-end tool is not without difficulties. First is the question of compatibility between the logic of the method and that of the back-end tool. The choices are between embedding the logic of the method in that of the tool, and simply replacing the former by the latter when generating proof obligations. Pratten [38] describes a tool that adopts the former approach: it generates a PVS representation of proof obligations for the B method that conform to the standard semantics of B given in [1]. The RSML table analyzer adopts the latter approach (which can also be considered a shallow embedding, since RSML specifications use a simple fragment of first order logic). Second is the issue of providing an adequate formalization of all the supporting theories required for a given specification. For example, formal analysis of a program that uses a data structure to represent a graph will require access to a formalization of some fragment of graph theory. If supporting theories are written in the notation of the formal method, then analysis will be complicated by their embedding into the language of the verification system; also, supporting theories should generally be written in a way that supports effective deduction (e.g., by presenting definitions and lemmas in a form that is convenient for rewriting), and this may be contrary to the style of the method. If the supporting theories are written directly in the language of the verification system, then the intended method is not followed to the full extent, and the specifier must master two different specification languages and styles.

Traditional shallow and deep embeddings also suffer from the drawbacks just outlined. Furthermore, the difficulties of embedding a formal specification language in a different logic are greater when the full notation is to be supported, rather than just its proof obligations. Agerholm [2] describes a shallow embedding of VDM-SL into PVS that transforms VDM-SL constructs to similar PVS constructs, and Agerholm, Bicarregui and Maharaj [3] describe an extension of this approach to support refinements. Although the constructs are often similar, they are not identical, so the semantics of the VDM-SL specifications are not fully preserved by this embedding. Agerholm and Frost [4] describe an alternative embedding of VDM-SL into Isabelle; here, the semantics are preserved but the embedding is correspondingly more difficult.

Whenever the notation of one method is supported by the logic and mechanization of another (whether as a back-end or by embedding), there is tension between supporting the semantics of the former vs. fully exploiting the mechanization of the latter. And if one notation is supported by more than one tool, there is the additional concern that each will provide slightly different semantics.

Structural embeddings sidestep these concerns because they do not claim to preserve the full semantics of the original method. A structural embedding of VDM, for example, would be similar to the first of the two VDM embeddings mentioned above, except that the logic of VDM would be *replaced* by that of the verification system concerned, and a traditional embedding would be provided only for the outermost, or structural level of the VDM language (e.g., its notions of state and of refinement). Of course, the resulting system would not support true VDM any more than PBS supports true B, and this would be a fatal defect for some users. However, we believe that others will value the methodological contributions of VDM, or B, more than the idiosyncrasies of their logics and would be happy to trade those logics for others in return for better automated support of their preferred method.

There are some potential difficulties, however, to this approach. In the first place, even quite good verification systems are not uniformly effective, and the encodings produced by structural embeddings may take them into areas where they perform poorly. For example, one of the proof obligations generated by the RSML checker caused PVS to go into an apparently endless computation [17] (this was a back-end application rather than a structural embedding but the problem would be the same in either case). In fact, PVS had discovered that the formula was not a propositional tautology within a couple of seconds (which is all the user wanted to know), and then spent the next several days trying to calculate a minimal set of subgoals to return to the user (there were well over 1,000). Design choices made in the expectation that the user is conducting an interactive proof of a human-generated conjecture may be inappropriate when dealing with formulas generated by mechanical translation.

A related problem is that most interactive verification systems assume that a human is guiding the process, and they therefore provide only rudimentary interfaces for other programs. A deeper manifestation of the same design philosophy is the monolithic, closed nature of most verification systems: it is almost impossible for outside programs to interact with their components or to query their internal data structures, and correspondingly difficult to create customized capabilities.

Our recommendation (which is hardly original) is that verification systems should be restructured into open collections of components with well-defined application programming interfaces (APIs) that allow other programs to invoke their capabilities. A cluster of components interacting through a shared intermediate language might be a suitable overall architecture.[4] A front-end providing structural embedding for some formal method could then communicate with the verification system through its intermediate language and its APIs.

Some embedding tools have already adopted a similar architecture, but with only monolithic verification systems connected to their intermediate languages. Gravell and Pratten [15] describe a tool that automates conventional embedding of a formal notation within the logic of a verification system. The tool, called JavaLIL, has been used for the embedding of Z specifications into the higher-order logics of PVS and HOL [14]. Gravell and Pratten justly bemoan difficulties caused by the monolithic, closed character of the verification systems used. In a similar vein, Jacobs et al. [20, 21] describe a tool called LOOP to support embeddings of object oriented languages in general-purpose verification systems.

Structural embedding does not serve the same ends as these tools: its purpose is not to support the full language of an existing formal method, but to capture just its *methodological* attributes and to support those in conjunction with the language of an existing verification system. We believe that those for whom methodology and mechanized support are more important than the authentic language of a specific formal method may find that a structural embedding provides a cost-effective way to achieve their goals.

Of course, structural embedding does not solve all the problems of providing effective automated support for formal methods. There is more to a method than just its deductive aspects (although deductive support is the *sine qua non* of truly *formal* methods): a fully supported method also supplies automated assistance in documentation and traceability, prototyping and code development, testing and validation, and the project management that ties all these together. We would hope that these capabilities could be created by customizing (or, if necessary, developing) generic tools that support these functions, and that such generic tools could be incorporated in the open architecture described previously.

Acknowledgments The authors would like to thank N. Shankar and the anonymous referees for constructive criticism and helpful comments.

[4] This is the approach adopted by the SAL (Symbolic Analysis Laboratory) project at SRI, Berkeley and Stanford. However, SAL is intended to promote cooperative use of complete tools such as model checkers and theorem provers, not the components of such tools; its focus is the use of abstraction in analysis of concurrent systems represented as transition systems.

References

1. J.-R. Abrial. *The B-Book—Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. Sten Agerholm. Translating specifications in VDM-SL to PVS. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 1–16, Turku, Finland, August 1996. Springer-Verlag.
3. Sten Agerholm, Juan Bicarregui, and Savi Maharaj. On the verification of VDM specification and refinement with PVS. In Juan Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT (Formal Approaches to Computing and Information Technology), chapter 6, pages 157–190. Springer-Verlag, London, UK, 1997.
4. Sten Agerholm and Jacob Frost. An Isabelle-based theorem prover for VDM-SL. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 1–16, Murray Hill, NJ, August 1997. Springer-Verlag.
5. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.
6. Michael J. Beeson. Towards a computation system based on set theory. *Theoretical Computer Science*, 60:297–340, 1988.
7. J.C. Bicarregui, D.L. Clutterbuck, G. Finnie, H. Haughton, K. Lano, H. Lesan, D.W.R.M. Marsh, B.M. Matthews, M.R. Moulding, A.R. Newton, B. Ritchie, T.G. A. Rushton, and P.N. Scharbach. Formal methods into practice: Case studies in the application of the B method. *IEE Proc. Software Engineering*, 144(2):119–133, 1997.
8. J.C. Bicarregui and B. Ritchie. Invariants, frames and postconditions: A comparison of the VDM and B notations. *IEEE Transactions on Software Engineering*, 21(2):79–89, February 1995.
9. J.-P. Bodeveix, M. Filali, and C. Muñoz. A formalization of the B-method in Coq and PVS. Manuscript, 1999.
10. R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.
11. M. Büchi. The B bank: A complete case study. In *Proc. ICFEM98*, pages 190–199. IEEE Press, December 1998.
12. Martin Büchi. The B bank. In Emil Sekerinski and Kaisa Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, FACIT (Formal Approaches to Computing and Information Technology), chapter 4, pages 115–180. Springer-Verlag, London, UK, 1999.
13. William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Rees. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
14. Andrew M. Gravell and Chris H. Pratten. Embedding a formal notation: Experiences of automating the embedding of Z in the higher order logics of PVS and HOL. In Grundy and Newey [16], pages 73–84. Available at `http://www.staff.ecs.soton.ac.uk/~amg/javalil/efn.ps.gz`.

15. Andrew M. Gravell and Chris H. Pratten. A prototype generic tool supporting the embedding of formal notations. In Grundy and Newey [16], pages 63–72. Available at `http://www.staff.ecs.soton.ac.uk/~amg/javalil/agt.ps.gz`.
16. Jim Grundy and Malcolm Newey, editors. *Theorem Proving in Higher Order Logics: Emerging Trends 11th International Conference, TPHOLs '98, Supplementary Proceedings*, Canberra, Australia, September 1998. Technical Report 98-08, Department of Computer Science, Australian National University.
17. Mats P. E. Heimdahl and Barbara J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *IEEE High-Assurance Systems Engineering Workshop (HASE '96)*, pages 252–262, Niagara on the Lake, Canada, October 1996.
18. Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
19. Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
20. Ulrich Hensel, Marieke Huisman, Bart Jacobs, and Hendrik Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Chris Hankin, editor, *Programming Languages and Systems: 7th European Symposium On Programming (ESOP)*, volume 1381 of *Lecture Notes in Computer Science*, pages 105–121, Lisbon, Portugal, March 1998. Springer-Verlag.
21. Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrick Tews. Reasoning about Java classes. In *Proceedings, Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 329–340, Vancouver, Canada, October 1998. Association for Computing Machinery. Proceedings issued as ACM SIGPLAN Notices Vol. 33, No. 10, October 1998.
22. C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.
23. Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.
24. K. Lano and H. Haughton. *Specification in B.* Imperial College Press, 1996.
25. Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.
26. Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 64–78, Palo Alto, CA, November 1996. Springer-Verlag.
27. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
28. C. Muñoz. PBS: Support for the B-method in PVS. Technical Report SRI-CSL-99-01, SRI International, February 1999.
29. University of Teesside. B-resource. Available at `http://www-scm.tees.ac.uk/bresource/welcome.html`.
30. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

31. Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.
32. David Y. W. Park, Jens U. Skakkebæk, Mats P. E. Heimdahl, Barbara J. Czerny, and David L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In Mark Ardis, editor, *Second Workshop on Formal Methods in Software Practice (FMSP '98)*, pages 34–43, Clearwater Beach, FL, March 1998. Association for Computing Machinery.
33. David Lorge Parnas. Tabular representation of relations. Technical Report CRL Report 260, Telecommunications Research Institute of Ontario (TRIO), Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, October 1992.
34. David Lorge Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.
35. David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.
36. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
37. Vaughan Pratt. Anatomy of the Pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, May 1995. Springer-Verlag.
38. C.H. Pratten. An introduction to proving AMN specifications with PVS and the AMN-PROOF tool. In Henri Habrias, editor, *Proc. Z Twenty Years On—What Is Its Future*, pages 149–165. IRIN-IUT de Nantes, October 1995.
39. H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 123–134, New Brunswick, NJ, July/August 1996. Springer-Verlag.
40. John Rushby. PVS bibliography. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA. Constantly updated; available at `http://www.csl.sri.com/pvs-bib.html`.
41. John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.
42. Mark Saaltink, Sentot Kromodimoeljo, Bill Pase, Dan Craigen, and Irwin Meisels. An EVES data abstraction example. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 578–596, Odense, Denmark, April 1993. Springer-Verlag.
43. Lance Sherry. Apparatus and method for controlling the vertical profile of an aircraft. United States Patent 5,337,982, August 16, 1994.
44. J.M. Spivey. *Introducing Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.