FROM VERIFIED SPECIFICATIONS TO VERIFIABLE SOFTWARE

Alwyn Goodloe†, Leonard Lensink‡, and César Muñoz§.

ABSTRACT

Declarative specifications of digital systems often contain parts that can be automatically translated into executable code. Automated code generation, as opposed to manual code writing, eliminates a potential source of errors when a prototype implementation of these models is required. Furthermore, code generation allows for better integration of formal methods into the software development process. However, for this approach to be effective, the generated code should be, ideally, as efficient as code that a normal programmer would write, and, more importantly, verifiable. We present a prototype code generator for the Prototype Verification System (PVS) that translates a subset of PVS functional specifications into an intermediate language and subsequently to multiple target languages. The generated code can be subjected to software verification tools such as verification condition generators, static analyzers, and software model-checkers, to increase the confidence that the generated code is correct. We illustrate this approach with the generation of verifiable Java code from the PVS specification of a verified distributed communication protocol.

1 INTRODUCTION

Formal methods refers to a set of mathematical notations and techniques employed in different phases of the development of a digital system with the goal of improving the quality of the system’s software (and hardware) artifacts. In its heavy-weight form, formal methods are used in the early stages of the software development cycle. In this case, mathematical models are written using formal specification languages [1,6,19,25,26,37,38,53] and properties of these models are verified using proof assistants and automated theorem provers. These models then serve as the basis for the development of the actual software, which is constructed via formal refinement, code synthesis, and manual coding.

Software verification is a lighter form of formal methods that typically focus on the final stages of the development cycle. In this case, software verification tools are used to construct abstract formal models of hand written code. These models are then checked against specific safety and liveness properties [13, 21, 22]. The heavier approach provides stronger guarantees but requires extensive human interaction. Furthermore, it leaves the question open to whether the constructed software actually conforms to the mathematical

---

*This work was supported by the National Aeronautics and Space Administration, Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043. Authors in alphabetical order.
†National Institute of Aerospace (NIA), 100 Exploration Way, Hampton VA, 23666. Email: Alwyn.Goodloe@nianet.org.
‡Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands. Email: l.lensink@cs.ru.nl.
§National Institute of Aerospace (NIA), 100 Exploration Way, Hampton VA, 23666. Email: munoz@nianet.org.
models. The lighter approach is highly automated, but, in practice, mostly effective for validation purpose.

As theorem provers become more and more powerful, verification condition generators (VCG’s) [4, 16, 23, 31] become a practical heavy-weight complement to software verification tools. VCG’s take as input programs that are annotated using logical assertions, and produce proof obligations that guarantee the correctness of the code with respect to these assertions. The proof obligations are then discharged, either manually or automatically, using theorem provers. This approach only works well as long as programmers write precise and relevant assertions.

Software testing can be enhanced by the existence of formal specifications by providing the test engineer a precise guide as to what the software is expected to do. In addition, formal specifications can drive the testing process further if automatic test-case generation is employed to create the test cases. For instance, generating unit-tests from Isabelle HOL specifications has been demonstrated in [2,10]. Sewell has demonstrated how formal specification can be used to drive the conformance testing [8] of TCP implementations. Software verification tools such as model checkers have also been used to generate counterexamples that, in turn, can serve as test cases to verify that the counterexamples do not occur in the implementation under test [3, 7, 12]. Additional techniques for generating test cases from labeled transition systems can be found in [45,49,50,52]. Hence, in addition to their primary role in verifying and validating software, formal methods and software verification can serve a support role in the software testing phase of the software development process.

In this paper, we are interested in automated code generation from formal specifications. Our work is related to heavy-weight formal methods in that our starting point is an already verified and refined formal specification. However, our research focuses on the error prone activity of producing high-quality code, for example in Java, that realizes a refined and verified specification, for example written in PVS. Our approach is also related to light-weight formal methods in that the generated code is annotated with formal annotations and therefore it is suitable for software verification tools such as VCG’s and software model checkers. These logical annotations are extracted from the original specification and, therefore, the approach does not depend on the ability of the programmer to write relevant annotations in the code. In cases where the generated implementation may serve as a reference implementation, the annotated code could be used to generate test cases.

We are mainly interested in safety critical software [44]. In particular, we focus on developing capabilities for handling fault-tolerant safety-critical systems for which there is a tradition of building formal models and mechanically proving that these models satisfy safety properties, for instance, correctness and completeness properties such as validity and agreement [33, 41]. Once these properties are verified, these models are implemented using traditional programming languages. A huge improvement to this scenario would be to have the code and the formal assertions derived from the proven models. This way, the possibility that errors are introduced during the coding phase is eliminated. This also enables the use of verification condition generators to check that the generated code is correct.

Generally, two different techniques are employed for generating code from formal specifications. The first technique exploits the Curry-Howard isomorphism in order to extract programs from constructive proofs [30, 40]. The second technique translates the original specification into code assuming that the specification has been sufficiently refined such that
it is written in an pseudo-executable subset of the specification language \[1, 25, 47, 53\]. The
latter technique is particularly appealing for generating code from specifications written in
declarative languages, such as PVS, since these languages encourage writing specifications
in a style that is in large part functional and, therefore, executable. Furthermore, in the
absence of constructive proofs, the second technique is usually the only option. For these
reasons, we decided to translate the executable subset of a declarative specification language
into code.

More precisely, we present a prototype generator of annotated code for declarative spec-
cifications written in the Prototype Verification System (PVS) \[38]\(^1\). We currently generate
Java code with assertions written in JML \[11\]. However, the code generator uses an interme-
diate language which enables multiple target languages. Our main goal is to generate code
that has the following properties:

- **Integrable:** The generated code should facilitate easy integration in existing and new
code bodies.

- **Efficient:** The generated code should reflect the structure of the original declara-
tive specification. However, it should use more efficient imperative features, such as
loops and side effects, whenever possible without compromising the correctness of the
generated code.

- **Verifiable:** The translation process should maintain the verified properties of the
original model. Elimination of errors introduced during a manual translation from
a model to a program should not be replaced with defects introduced by the code
generation.

The rest of this paper is organized as follows. Section 2 gives an overview of the proposed
approach and briefly describes the tools on which our work relies. Section 3 is a survey of
the methodology employed to translate PVS into Java code. In Section 4, we present a case
study taken from a verified distributed communication protocol. Section 5 discusses related
works. The last section presents future work and final remarks.

2 FROM PVS TO JAVA AND BACK AGAIN

The input to our code generator is a declarative specification written in PVS, a higher
order logic specification language and theorem prover \[38\]. Since we aim at a wide range of
applications, we do not fix the target language. Indeed, the tool first generates code in Why,
an intermediary language for program verification \[17\]. Our current prototype generates
Java annotated code from the Why code. In the future, we may implement outputs for
other functional and imperative programming languages.

In addition to enabling multi-target generation of code, another benefit of an intermediate
language is that transformations and analysis that are independent from the target language
can be applied to the intermediate code directly. This saves a lot of optimizations that
otherwise have to be performed for every target language. Furthermore, our code generator
exports Why code into XML. This liberates the developer of the translation to a specific

\(^1\)The prototype is electronically available from http://research.nianet.org/fm-at-nia/PERICO/.
target language from delving into the internals of the generator or having to write a custom parser. Any modern programming language provides an off-the-shelf XML parser.

In order to increase the confidence on the generated code, the generator annotates the code with logical assertions such as pre-, post-conditions, and invariants. These assertions are extracted from the declarations, definitions, and lemmas in the formal model. Therefore, the generated code can be the input of a verification condition generator, in our case Krakatoa [16]. Krakatoa generates proof obligations for several theorem provers, including PVS. We remark that the generated PVS proof obligations are different from the original PVS specification. However, if the original specification has been shown to be correct, discharging the proof obligations is a relatively easy task. The annotated code is also amenable to static analysis, software model checking, and automated test generation.

The proposed approach is illustrated by Figure 1, where the dashed line encloses the functionality currently implemented in our prototype. The rest of this section gives a short overview of PVS and Why.

2.1 PVS

PVS is an interactive environment for writing formal specifications and for checking proofs. It contains an expressive specification language and a powerful theorem prover. It has been applied to large applications both in academic as well as industrial settings [39].

The specification language is based on classical higher order logic, augmented with a sophisticated type system. The type system uses predicate subtypes and dependent types. It also has a mechanism for defining algebraic data types. All functions that are defined in the specification language must be total. However, partial functions can be defined by
restricting the domain of the function to a subtype using predicate subtyping. The many
features of the PVS type system make it very powerful, but also makes type checking in
general undecidable. The theorem prover generates type correctness conditions (TCC’s) for
the undecidable parts of the type checking mechanism. In practice, most of the TCC’s are
automatically discharged by the system.

The theorem prover is used either interactively or in batch mode. The basic deductive
steps range from small inferences to use of decision procedures for, among others, arithmetic
reasoning and propositional simplification. Using a scripting language, these basic steps can
be built into larger procedures. The proof checker manages the proof construction by asking
the user to supply a proof command that will either prove the current goal or generate one
of more new goals. Once all goals have been reached, the theorem is considered proven.

2.2 Why

Why is a multi-target verification condition generator developed by Jean-Christophe Filliâtre
et al. [15, 16, 18]. The Why tool generates proof obligations for different kinds of existing
proof tools, including proof assistants such as PVS but also automated first order theorem
 provers and SMT solvers.

Why builds a functional interpretation of the imperative program given as input. This
interpretation contains both a computational and a logical part. Using this information,
the tool applies a Hoare logic and Dijkstra’s calculus of weakest preconditions to generate
proof obligations. Why’s input language, which is also called Why, is based on ML and
has imperative features, such as references and exceptions, and functional features, such as
higher-order functions. In contrast to ML, aliasing between mutable variables is not allowed.
This constraint is guaranteed by the typing rules of the Why language [15].

The Why tool is used as the back-end of verification condition generators. Indeed, the
same team that develops Why, develops the tools Krakatoa and Caduceus, which are front-
ends for Java and C VCG’s, respectively. Figure 2 illustrates the Why/Krakatoa/Caduceus
tools suite.

3 CODE GENERATION

A direct translation from a PVS specification into the Why language is fairly straightforward.
Each language construct in the executable subset of the PVS specification language has an
almost immediate counterpart in the Why language. Indeed, like PVS, Why can be used as
a purely functional programming language.

Since our final goal is to produce efficient and high-quality annotated code, we have to
carefully consider how each PVS construct is translated into Why. In some cases, we had to
extend the Why core language with new features.

PVS specifications are structured in theories, which are collections of declarations, defini-
tions, and formulas. Theories can be parametrized by types and mathematical objects. The
Why language does not have a similar kind of construct. As result, a direct translation of a
PVS specification into Why would result in a flat monolithic code. Hence, we have added to
Why a simple notion of parametric modules. These modules provide a naming scope for a
set of Why declarations and allow for modularity in the generated programs. We note that a
more general notion of module that includes the notion of interface is currently begin added
to the Why core language [51].
The PVS specification language is richer than the Why language. For example, PVS supports records, tuples, and abstract data types. In this case, we decided not to modify the core Why language, but to treat these constructs as syntactic sugar in such a way that the final code retains the syntactical similarity to the original PVS specification.

One of the main concerns in generating code from a declarative specification is the efficiency of the resulting program. Purely functional programs are not terribly efficient. The obvious difference between PVS and Why is that Why supports imperative features such as references and side effects. The efficiency of the generated Why code could be significantly improved if some PVS constructs are translated into imperative Why code. For instance, PVS supports record and array overriding, e.g., if $A$ is an array of integer values, the PVS expression $A$ WITH [(0) := 10] denotes an array that is equivalent to $A$ in all indices except 0 where it has the value 10. It is particularly tempting to translate the PVS overriding feature using destructive updates in Why: $A[0] := 10$. In this case, the index 0 of the array $A$ is set to 10. Why destructive updates are more efficient than PVS purely functional overriding. However, as we will see in Section 3.1, a careful analysis has to be performed to guarantee the correctness of this translation.

An important difference between the PVS and the Why logical frameworks is that Why distinguishes between logical and computational values. For example, the PVS function

\[
is\_square(x, y: real): bool = (y*y=x)\]

defines a function that returns true when the second argument is the square root of the first argument. This function can be translated into Why as a proposition:
\textbf{predicate} \textit{is\_square}(x:real, y:real) = (y*y=x)

which can be used in logical assertions. The PVS function \textit{is\_square} can also be translated as a \textit{computational element} in Why:

\textbf{let} \textit{is\_square}(x:real, y:real):bool
    y*y=x;

which can be used in programs. The same distinction applies to general functions which can be defined as logical functions, to be used in propositions, or as programs.

Since the set of propositions and programs is disjoint, e.g., propositions cannot appear in programs and programs cannot appear in propositions, we have to generate the appropriate Why code depending on how PVS expressions are being used in the formal model. We discuss in Section 3.2 which part of a PVS specification is used to generate logical assertions.

The last step of our approach is the translation of Why programs into the target language. Section 3.3 provides a detailed description of the generation of actual Java code.

\section{3.1 Destructive Updates}

The PVS ground evaluator includes a highly efficient code generator that translates PVS expressions into Lisp \cite{46}. In the generated Lisp code, a PVS overriding expression is translated into two variants. One that destructively updates the data structure and one that constructs a new copy. If the variable being overridden cannot be aliased, the destructive version is chosen, while if there is a risk of aliasing, the safe version that makes a copy is used. The analysis is a conservative approximation. Nested applications and higher order operations will make the alias analysis difficult or even impossible. Furthermore, the PVS ground evaluator does not attempt to transform the program in order to make possible a more accurate analysis.

Consider the following PVS function that negates each element of an array of 1000 elements:

\begin{verbatim}
Arr : TYPE = ARRAY[below(1000) \rightarrow int]
negate(A:Arr, i:below(1000)) : RECURSIVE Arr =
    IF i=0 THEN A
    ELSE negate(A \WITH [(i-1) := -A(i-1)], i-1)
ENDIF
MEASURE i
\end{verbatim}

The destructive update of the array \textit{A} can be done safely because the update of element \textit{i-1} is done after the value of the element has been read and there is no reference to \textit{i-1} afterward. If the update is done non-destructively, an array copy would have to be performed 1000 times with disastrous performance results. On the other hand, since it is not always possible to use the destructively updating variant of the function \textit{negate}, the translator generates a destructive and a non-destructive version of each function that updates a variable.

The translation from PVS to Why uses a similar mechanism. However, due to the aliasing exclusion mechanism built into the type system of Why, it is impossible to directly translate array updates in PVS into their counterparts with references in the Why language. In our case, we only generate a destructive variant of every function. If the alias analysis determines that a particular variable cannot be destructively updated, we create of deep
copy of this variable before performing any destructive update. Hence, we safely perform the computation without destroying the initial object and we avoid the possible introduction of aliasing.

### 3.2 Assertions

The predicate subtyping capability of PVS allows for the precise specification of functions akin to pre- and post-conditions in traditional Hoare logic-based specification languages [43]. For instance, the square root function in PVS can declared as follows:

\[
\text{sqrt}(x: \text{real} \mid x \geq 0) : \{y: \text{real} \mid y \geq 0 \land x = y*y\}
\]

This declaration states that \texttt{sqrt} is a function that takes a non-negative real \(x\) and returns a non-negative real \(y\) such that \(x=y*y\).

The PVS to Why generator uses the type information of PVS declarations to extract pre-and post-conditions for the Why version of these declarations. In the particular case of recursive declarations, the type of the arguments are extracted as invariants, and the measure information is extracted as the termination argument. Furthermore, functions used in type definitions are extracted as logic functions rather than programs. This overcomes the Why restriction that forbids the use of programs in logical statements.

The Why language does not have the means to specify proofs. However, it allows for the specification of axioms, which are used by automated theorem provers to discharge the proof obligations. We translate all the lemmas and TCC’s into axioms in the Why logic.

### 3.3 Generation of Java

This section describes the generation of Java code. As explained in previous sections, the translations from PVS to Java uses the intermediate language Why. However, to emphasize the syntactical relation between the source language, i.e., PVS, and the target language, i.e., Java, we illustrate this translation with examples of PVS constructs and the resulting Java code without providing the intermediate representation in Why. The examples are taken from the case study given in Section 4.

#### 3.3.1 Primitive types

The PVS specification language contains the normal range of primitive types: \texttt{nat}, \texttt{number}, \texttt{boolean}. Although usually included as a primitive type, chars and strings are not primitive types in PVS. Instead they are modeled as ASCII numbers and finite sequences of characters respectively. The immediate counterparts of the PVS primitive types are in Java: \texttt{int}, \texttt{float} and \texttt{boolean}. At this point, we are ignoring the obvious difference between PVS’s \texttt{real} and Java’s \texttt{float}. This difference may be addressed in the future as the PVS NASA Libraries includes a specification of the IEEE 854 Floating Point standard [9,32].

#### 3.3.2 Subtypes

In PVS, types can be much more expressive than what is common in a programming language and are one of the prime methods of specifying the properties of a model. For instance, it is possible to specify that the argument of a function is a natural number, but also, that the number has to be in a certain range.
For example, the PVS declaration

```
length : upto(maxsize)
```

restricts length to a natural number (the type of `maxsize`) upto `maxsize`. In most programming languages, there is no direct support for these kinds of types, so all subtype declarations are translated into its supertype:

```
int length;
```

As discussed in Section 3.2, we use this type information to generate annotations to Java that will help in ensuring that the generated code is indeed correct. In Section 3.3.13, a more detailed explanation of the method to generate JML assertions is given.

### 3.3.3 Records

Records are one of the most widely used structures in PVS specifications. For instance, a state space is often encoded as a record. Records are translated into Java classes, where each field of the record corresponds to an attribute of that class in Java. For instance, the record

```
WinReceiverPrivate : TYPE = [#
  nd : nat,
  % nd = Next to be Delivered
  % Msgs below nd have been delivered
  % nd is base of receiver window

  la : upto(nd),
  % Last to be Acknowledged
  % Msgs below la have been delivered and acked

  lr : subrange(nd,nd+rw),
  % lr = Last Received
  % Msgs upfrom lr have NOT been received.

  rcvd : (max_window?(lr-nd))
  % Receiver window
#
```

translates into the following java class:

```
static public class WinReceiverPrivate implements Cloneable {
  int la;
  int lr;
  int nd;
  WindowTheory.Window rcvd;

  public WinReceiverPrivate(int la
    ,int lr
    ,int nd
    ,WindowTheory.Window rcvd) {
    this.la = la;
    this.lr = lr;
    this.nd = nd;
    this.rcvd = rcvd;
  }

  public WinReceiverPrivate update(int la
    ,int lr
    ,int nd
    ,WindowTheory.Window rcvd) {
    this.la = la;
    this.lr = lr;
    this.nd = nd;
    this.rcvd = rcvd;
  }
}
```
public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        return this;
    }
}

Note that the subtypes of the individual fields are translated into their supertypes in Java. The supertype of the \texttt{rcvd} field is a record that is defined in a different theory.

Besides the attributes, the generated Java class also contains other functions.

- First of all, a constructor that will instantiate the particular record with the values that are passed to it. This allows the creation of new records when a record literal is encountered in the PVS specification.

  Consider the following example:

  \begin{verbatim}
  init_win_receiver : WinReceiverPrivate = nd := 0, la := 0, lr := 0, rcvd := init
  \end{verbatim}

  This record literal, which initializes the state of the receiver to a starting state is translated into java by means of a function call to the constructor. A new record will be instantiated and returned to the caller.

  \begin{verbatim}
  public WinReceiverPrivate init_win_receiver() {
      return new WinReceiverPrivate(0, 0, 0, windowtheory.init());
  }
  \end{verbatim}

- Secondly, an update function is generated that will perform destructive updates of the record if it is determined to be allowed. Updates in PVS are done using the \texttt{WITH} operator. Consider the case of the \texttt{WinReceiverPrivate} variable \texttt{priv} defined as follows:

  \begin{verbatim}
  priv WITH [ nd := priv'nd + ff, 
              lr := max(priv'lr, idx+1), 
              la := priv'la, 
              rcvd := slide(nrcvd, ff) ]
  \end{verbatim}

  Using the generated update function this is straightforwardly translated to:
priv.update( priv.la
    , Math.max(priv.lr, idx+1)
    , priv.nd++
    , windowtheory.slide(nrcvd, ff));

- Thirdly, the clone() function is generated to make copies of existing records. This is needed because using a destructive update on a variable is not always safe. It is not allowed if an updated variable is referenced later. In PVS, a variable has a static semantics. This will clearly not be the case if a destructive update is used. An analysis, explained in Section 3.1 is used to determine if a destructive update can be used or whether a copy should be made.

- Finally, we must include the capability to translate arrays of records. In order to provide such support, a function is generated within the class declaration to construct an array of records of a certain size. A initialization function for the array can be passed to this function.

```java
public static WinReceiverPrivate[]
    new_WinReceiverPrivate( int size,
        Lambda<Integer,WinReceiverPrivate> lambda) {
        WinReceiverPrivate[] arrayWinReceiverPrivate =
            new WinReceiverPrivate[size];
        for (int i=0; i<size; i++)
            arrayWinReceiverPrivate[i] = lambda.curry(i);
        return arrayWinReceiverPrivate;
    }
```

### 3.3.4 Tuples

At the moment there is no support for the translation of tuple types to Java. Implementation is fairly easy since semantically they are equivalent to records where each field is denoted by the position of each element in the tuple.

### 3.3.5 Arrays and Function Types

In PVS, arrays and functions are identical types. An array is interpreted as a function from the natural numbers to a certain range. For efficiency reasons, translating functions that act as an array within the specification to functions is not a good idea. On the other hand, what is to be perceived as an infinite array within the specification cannot be directly translated into an array. Therefore, we create an array in Java if the domain of the function is finite and a (higher order) function if the domain cannot determined to be finite. For instance, in the example that follows, Window contains a record definition that contains a pair of arrays for both the mask and data fields.

```plaintext
Window : TYPE = [#
    length : upto(maxsize),
    mask   : { seq : ARRAY[below(maxsize)→bool] | ...
```
\[ \forall (i:\text{subrange}(\text{length},\text{maxsize}-1)) : \neg \text{seq}(i) \}, \]

\[ \text{data} : \text{ARRAY[below(\text{maxsize})\rightarrow Data]} \]

Since \text{Window} is a record, a class is generated. Because the mask and data fields are declared as function types with a finite domain due to the \text{below(\text{maxsize})}, both fields are declared as Java arrays.

```java
static public class Window implements Cloneable {
    int[] data;
    int length;
    boolean[] mask;

    public Window(int[] data, int length, boolean[] mask) {
        this.data = data;
        this.length = length;
        this.mask = mask;
    }
}
```

Depending on the context of the update, an array update will either be translated into a predefined function that updates an array value, or into a common array update. For instance, \( x \text{ WITH } [ (n) := \text{data} ] \) can be translated into either \text{Prelude.update}(x, n, \text{data}) or \( x[n] = \text{data}; \). The first translation will be used when the update occurs on an expression position like happens for instance in the \text{SlidingWinReceiver} class:

```
LET x1 = x \text{ WITH } [ (n) := \text{data} ] \text{ IN } [...]
```

Will be translated to

```
final int[] x1 = Prelude.update(x, n, \text{data});
```

The latter translation will only be used if the update occurs on a position that allows for a translation into a java statement instead of an expression.

### 3.3.6 Abstract Datatypes

Most of the commonly used abstract datatypes are supported in PVS. Any abstract datatype that can be recursively generated using constructors is supported. This includes datatypes like trees and lists, but, for instance, not bags.

Consider the case of a datatype definition that is composed of constructors \text{DataFrame} and \text{AckFrame}, recognizers \text{isDataFrame} and \text{isAckFrame} and accessors \text{index}, \text{data}, \text{lb} and \text{ub}.

```pvs
WindowFrame : DATATYPE
BEGIN
    DataFrame(index:nat, data:Data) : isDataFrame % Regular data
    AckFrame(lb:nat, ub:upfrom(lb)) : isAckFrame % Acknowledgment
END WindowFrame
```

The translation to java uses a class to store all the information of the abstract datatype. In case of a recursively defined datatype, one of the attributes of the class will be of the same type as the class.
public class WindowFrame {
    boolean isDataFrame = false;
    int index;
    int data;
    boolean isAckFrame = false;
    int lb;
    int ub;
}

All the constructors, recognizers and accessors are generated as functions.

public WindowFrame DataFrame(int index, int data) {
    WindowFrame WindowFrame = new WindowFrame();
    WindowFrame.isDataFrame = true;
    WindowFrame.index = index;
    WindowFrame.data = data;
    return WindowFrame;
}

public WindowFrame AckFrame(int lb, int ub) {
    WindowFrame WindowFrame = new WindowFrame();
    WindowFrame.isAckFrame = true;
    WindowFrame.lb = lb;
    WindowFrame.ub = ub;
    return WindowFrame;
}

public int index(WindowFrame WindowFrame) {
    return WindowFrame.index;
}

public int data(WindowFrame WindowFrame) {
    return WindowFrame.data;
}

public int lb(WindowFrame WindowFrame) {
    return WindowFrame.lb;
}

public int ub(WindowFrame WindowFrame) {
    return WindowFrame.ub;
}

public boolean isDataFrame(WindowFrame WindowFrame) {
    return WindowFrame.isDataFrame;
}

public boolean isAckFrame(WindowFrame WindowFrame) {
    return WindowFrame.isAckFrame;
}

Not shown, but also generated are higher order versions of these functions.
3.3.7 Enumerations

Enumeration types are supported by taking advantage of the fact that they are internally represented as abstract datatypes.

Consider the following definition of an enumerated type:

\[
\text{WinReceiverAction} : \text{TYPE} = \{ \text{Receive, SendAck} \}
\]

This is equivalent to the abstract datatype:

\[
\text{WinReceiverAction} : \text{DATATYPE} \\
\begin{align*}
\text{Receive} : & \text{isReceive} \quad \% \text{Regular data} \\
\text{SendAck} : & \text{isSendAck} \quad \% \text{Acknowledgment}
\end{align*}
\]

The mechanism that generates java for abstract datatypes can subsequently be invoked. Java supports enumeration types since version 1.6. A future enhancement will be to make use of this native support.

3.3.8 Generic Types

It is possible to create generic types in PVS by parameterizing the type definition. Consider the following definition of a generic fifo queue parameterized by a type \( T \) that may be instantiated with any type.

\[
\text{fifoTheory}[\ T \ : \ \text{TYPE} \] : \text{THEORY} \\
\begin{align*}
\text{fifo} : & \text{TYPE} = [\# \text{lenth:nat, seq:ARRAY[nat} \rightarrow T] \#] \\
\text{nfx} : & \text{VAR} \ [ \text{nat} \rightarrow T] \\
\text{topof}(\text{nfx}) : & T = \text{nfx}(0)
\end{align*}
\]

The same functionality can be achieved in Java with generics, supported since JDK 1.5. The generic theory will be translated into a generic class in java. The type parameter in Java will be used at all the places it is used in the original PVS specification.

\[
\text{public class fifoTheory<T> {}
\begin{align*}
\text{public class fifo implements Cloneable} \{ \\
\text{int} & \text{lenth;} \\
\text{T[]} & \text{seq;}
\end{align*}
\]

\[
\text{public fifo(int lenth,T[]} \text{seq)} \{
\text{this.lenth = lenth;}
\text{this.seq = seq;}
\}
\]

14
public fifo update(int length, T[] seq) {
    this.length = length;
    this.seq = seq;
    return this;
}

public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        return this;
    }
}

public T topof(final T[] nfx) {
    return nfx[0];
}

3.3.9 Control Structures

Being functional in nature, PVS does not support the wide array of control structures of an imperative language. The most important control structure is conditional choice using an IF .. THEN .. ELSE construction.

\[
\text{LET } x = w.'\text{mask in}
\]
\[
\text{If } n < w.'\text{length } \land \ x(n) \text{ THEN}
\]
\[
\text{best_prefix}(w, n+1)
\]
\[
\text{ELSE } n
\]
\[
\text{ENDIF}
\]

The translation is very straightforward.

final boolean [] x = w.mask;
if (n < w.length && x[n]) {
    return best_prefix(w, n+1);
} else {
    return n;
}

Slightly more intricate is the CASES case distinction. This allows for simple pattern matching on constructors of abstract datatypes. Although not used in our model, pattern matching is simulated in Java using local variables and the java if(..) { .. } else { .. } construct.
3.3.10 Quantifiers

In general, logical quantification cannot be translated. However, if the domain is finite, universal and existential quantifiers are translated into loops that will either test for all values or test until a true value has been obtained.

Consider the following example:

% n is a prefix of window w if all masks below n are true
\( \text{isprefix}(w: \text{Window}, n: \text{upto}(w' \text{length})) : \text{bool} = \)
\[ \text{LET } x = w' \text{mask IN} \]
\[ \forall (i: \text{below}(n)) : x(i) \]

The \(\forall\) quantifier is translated into a \texttt{Prelude.forall} function call.

\begin{verbatim}
public boolean isprefix(final Window w, final int n) {
    final boolean[] x = w.mask;
    return Prelude.forall(0, n−1, new Lambda<Integer,Boolean>() {
        public Boolean curry(final Integer i) {
            return x[i];
        }
    });
}
\end{verbatim}

Where \texttt{Prelude.forall} is defined as a loop over the finite domain that returns \texttt{false} if the predicate, that is passed as a higher order function, fails. Otherwise \texttt{true} is returned.

\begin{verbatim}
public static boolean forall(int lb, int ub, Lambda<Integer,Boolean> lambda) {
    for (int i=0;i < ub−lb;i++)
        if (!lambda.curry(i))
            return false;
    return true;
}
\end{verbatim}

The \(\exists\) quantifier is defined analogously.

3.3.11 Functions

Most function definitions can easily be translated into functions in Java. Consider the following example:

% Window of length n
\( \text{iswindow}(n: \text{upto}(\text{maxsize}), w: \text{Window}) : \text{bool} = \)
\[ w' \text{length} = n \]

The translated Java function will be almost identical:

\begin{verbatim}
public boolean iswindow(final int n, final Window w) {
    return w.length == n;
}
\end{verbatim}

The main challenge in translating functions from PVS to Java is that the former is a functional specification language and allows higher order function. In Java, higher order
functions do not exist. This means that functions cannot be passed as arguments. However, Java does have objects that can be passed as arguments.

These objects can act as a wrapper to the functions that need to be passed. We create a special `Lambda` class:

```java
public abstract class Lambda<T1, T2> {
    abstract public T2 curry(T1 obj);
}
```

This parametrized class demands that on instantiation an anonymous inner class is passed that defines the curry function. This curry function will call the original function. For all functions that are defined, a higher order version is automatically generated.

```java
// Higher order function iswindow
public Lambda<Window, Lambda<Integer, Boolean>> iswindow
    = new Lambda<Window, Lambda<Integer, Boolean>>() {
    public Lambda<Integer, Boolean> curry(final Window w) {
        public Boolean curry(final Integer n) {
            return iswindow(n, w);
        }
    }
};
```

As can be seen in this example for the automatically generated higher order version of the `iswindow` function, the function is automatically currified. Thus for each of the variables of the original function, an inner function is created that applies a parameter and returns a new function which subsequently can be applied to next variable.

Some functions are naturally defined as higher order in PVS. Consider the example given below that taken from a theory that defines a bag.

```pvs
remove_if(p: [LinkFrame => boolean], b: bag)(e: LinkFrame): nat =
    IF p(e) THEN 0
    ELSE b(e)
ENDIF
```

Using the Lambda class, this is translated into

```java
public Lambda<LinkInterfaceTheory.LinkFrame, Integer> remove_if(
    final Lambda<LinkInterfaceTheory.LinkFrame, Boolean> p,
    final Lambda<LinkInterfaceTheory.LinkFrame, Integer> b) {
    return new Lambda<LinkInterfaceTheory.LinkFrame, Integer>() {
        public Integer curry(final LinkInterfaceTheory.LinkFrame e) {
            return (p.curry(e) ? 0 : b.curry(e));
        }
    };
}
```

Anonymous functions, or lambda expressions are translated similarly.

These kinds of translations are not the most efficient possible. In a functional language compiler, closures and higher order function calls are typically removed by transforming the program into a program in continuation passing style. This would however destroy the structure and readability of the transformed program.
3.3.12 Modules

The PVS specification language allows for a modular organization of all the theories. Each theory can import other theories which exposes the definitions of the imported theory. It also exposes the declarations that are visible to the imported theory if it is using an import statement as well.

Theories can be parameterized by regular parameters or by type parameters.

```pvs
WindowFrameTheory[ rw : posnat ] : THEORY
BEGIN
  IMPORTING WindowTheory[rw]
  [..]
END WindowFrameTheory
```

PVS theories are translated into Java classes. The parameters are translated into attributes of that class. If the parameter is a type variable, the class will be a generic class, as discussed in Section 3.3.8.

Whenever a class is imported, it is instantiated using the parametrized constructor.

```java
public class WindowFrameTheory {

  int rw;
  WindowTheory windowtheory;

  public WindowFrameTheory(int rw) {
    this.rw = rw;
    windowtheory = new WindowTheory(rw);
  }

  In this case, the class WindowFrameTheory is parametrized with rw and it passes this parameter on to the imported class WindowTheory.

3.3.13 Assertions

Since Java 1.4 it is possible to use asserts to force the java virtual machine to check assumptions about a program. This mechanism is limited in its expressiveness, so we have opted to use the extended possibilities that a markup language like JML offers.

In JML, one can specify pre- and postconditions and invariants. One of the design goals was ease of use Java programmers. The specifications are written as Java comments and the properties are specified as a Java boolean function.

The (sub)-type information present in the PVS model is extracted and translated into JML assertions. Whenever a type definition is encountered, a boolean function is generated that checks whether a given argument satisfies the properties of the subtype.

For instance, all the indices of the arrays in the verified distributed communication protocol are bounded by a maximum size.
uptoMaxSize : TYPE = upto(maxsize)

belowMaxSize : TYPE = below(maxsize)

These definitions are translated into boolean functions that check whether the argument, which is translated to its supertype in Java, satisfies the restrictions imposed on it by the subtype definition.

```java
public boolean uptoMaxSize(final int x) {
    return Prelude.nat(x) && Prelude.upto(x, maxsize());
}

public boolean belowMaxSize(final int x) {
    return Prelude.nat(x) && Prelude.below(x, maxsize());
}
```

The prelude functions `nat`, `upto` and `below` are defined as one would expect.

```java
public boolean nat(final int x) {
    return x >= 0;
}

public boolean upto(final int x, final int r) {
    return x <= r;
}

public boolean below(final int x, final int r) {
    return x < r;
}
```

Record type definitions can have type restrictions on their fields, so any recordtype definition generates a boolean function as well.

```java
Window : TYPE = [#
    length : uptoMaxSize,
    mask : { seq : ARRAY[belowMaxSize→bool] |
        ∀ (i : subrange(length, maxsize-1)) : ¬seq(i) },
    data : ARRAY[belowMaxSize→Data]
#]
```

Java already typechecks whether an object is a `Window` class, so the only checks that have to be added are to verify that the fields are satisfying their types.

- This means that the `length` field of the `Window w` must satisfy `uptoMaxSize`.
- The set-like notation for the type of the `mask` field defines a restriction on the subtype `ARRAY[belowMaxSize→bool]`. Besides `belowMaxSize(w.mask.length)`, the set predicate must also hold for `w.mask`.
- The data field again has only a simple bounds restriction.

```java
public boolean Window(Window w) {
    return uptoMaxSize(w.length)
    && (belowMaxSize(w.mask.length) &&
```
Prelude.forall(w.length, maxsize() - 1, new Lambda<Integer, Boolean>(){
    public Boolean curry(final Integer i) {
        return !w.data[i];
    }
})
&& belowMaxSize(w.mask.length);
}

Although JML supports a forall statement, at the moment we use the translation we already have for the PVS \forall statement. These generated boolean functions are then used as a predicate on the types of the functions that define the model.

% Window of length n
iswindow(n:upto(maxsize), w:Window) : bool =
    w.length = n

For instance, the function that checks whether a window is of a certain size is translated into a common Java function with the precondition that the parameters satisfy the generated functions for a Window recordtype and an uptoMaxSize type. The result of the function must be a boolean, the returntype of the function.

//@ requires uptoMaxSize(n) && Window(w)
//@ ensures Prelude.boolean(!result)
public boolean iswindow(final int n, final Window w) {
    return w.length == n;
}

4 CASE STUDY

In this section, we shall present a case study of a sliding-window protocol with block reply called the Guaranteed Delivery Protocol (GDP). This protocol was originally designed to serve in a protocol stack intended to facilitate communication between a remotely piloted vehicle and a ground based pilot; a detailed description of the entire system can be found in [35]. We begin with an overview of the protocol, this is followed by a presentation of a fragment of the PVS specification of the GDP receiver process. Many of the structures used as examples in Section 3 were taken from this specification. We then present the Java code of the PVS fragments in question. Finally we present some statistics about the generated code and discuss invariants that may be proved using software verification tools.

The GDP protocol assumes that at the sender node, data to be sent has been placed in the Input queue of type Data and at the receiving node, the protocol places data in the Output queue of type Data. The communication medium between sending and receiving node is called the ether, which is modeled as a pair of fifo queues of type WindowFrame (Section 3.3.6) representing channels flowing in each direction. The basic structure is shown in Figure 3.

The architecture upon which the protocol runs is assumed to be a distributed system composed of a sender node and receiver node communicating via our protocol. In practice, both nodes possesses the GDP sender and receiver capabilities, but for illustrative purposes we consider the sender functionality only present on the node sending data and the receiver
functionality only being present on the node receiving data. The system configuration is illustrated by Figure 4.

We now give a more detailed description of the GDP protocol’s behavior. As with all sliding-window protocols, GDP exhibits the following characteristics:

- Each message is assigned a sequence number that acts as an identifier.
- The protocol receiver process acknowledges the receipt of data messages by sending an acknowledgment message to the sender.
  - If the sender has not received an acknowledgement that a message has been received in a predefined time, then a timeout will occur and the protocol will resend that message.
  - If a receiver has already received and acknowledged a message, but the same message (defined as having same sequence number) is received again, then the system will resend the acknowledgement, but nothing is done with the data since it has already been processed. This covers the situation where an acknowledgement message is lost or corrupted in transit and the sender resends a message.
- There is an upper bound $sw$ on the number of data messages that can be sent without receiving acknowledgment for any of them. There is also an upper bound $rw$ on the
The number of data messages that can be received without sending an acknowledgment. The value of \( rw \) should be chosen so that \( rw \leq sw \). The value \( sw \) is called the sender’s window size and the value \( rw \) is called the receiver’s window size.

The protocol sender maintains a bounded buffer called \( \text{ackd} \) of type \( \text{Window} \) (Section 3.3.3). This buffer acts as a “window” of data that has yet to be sent as well as data that has been sent but not yet acknowledged. The buffer index indicates the sequence order in which messages are sent. The sender also maintains a collection of pointers. The variable \( ns \) is a pointer to the sequence number of the next data item to be sent and the variable \( na \) is a pointer to the first sequence number that has yet to be acknowledged. That is, sequence numbers \( 0, \ldots, na - 1 \) have all been acknowledged as received by the sender, but sequence number \( na \) has not yet been acknowledged. The buffer and pointers are kept in the structure \( \text{WinSenderPrivate} \).

The receiver also maintains a bounded buffer \( \text{rcvd} \) containing data received from the sender, but not yet moved to the \text{Output} queue. It also maintains a collection of pointers. The data is indexed by the sequence numbers. The variable \( la \) points to the last acknowledged sequence number, i.e., acknowledgment messages have been sent for sequence numbers \( 0, \ldots, la - 1 \). The variable \( nd \) points to the lowest sequence number that has yet to be delivered to the \text{Output} queue. The variable \( lr \) points the highest sequence number that has been received and \( lr \leq nd + rw \). The receiver ignores messages with sequence numbers greater than \( nd + rw \). When the receiver has received the contiguous block of sequence numbers \( nd, \ldots, x \), the pointer \( nd \) is slid forward to \( x + 1 \). Note that messages \( la, \ldots, nd - 1 \) have been received, but not yet acknowledged. Periodically, GDP sends an acknowledgment message that acknowledges the receipt of messages \( la, \ldots, nd - 1 \) and \( la \) is reset to \( nd \). The buffer and pointers are kept in the structure \( \text{WinReceiverPrivate} \) (Section 3.3.3).

We now consider the specification of the GDP receiver process. The receiver can perform one of two actions: it can process the receipt of a message or send an acknowledgement. This is reflected in the definition of the enumerated type \( \text{WinReceiverAction} \) in Section 3.3.7. The state machine for the receiver is defined as a function mapping the current state \( s \) and a specified action \( a \) to the next state \( n \). The action is determined nondeterministically as

\[
\text{GDPReceiver}(s, n : \text{GDPReceiver}) = ( \exists a : \text{WinReceiverAction}. n = \text{GPDReceiverNext}(s, a)).
\]

A fragment of the PVS specification of the state machine for the receiver’s next function is given below. Recall that the protocol was originally designed to function as part of a larger protocol stack. There is the presumption of the existence of a link layer, but in context of this investigation, GDP is directly connected to the ether, although, the names of the queues \( \text{link\_input} \) and \( \text{link\_output} \) reflect the original intention. The PVS fragment presented is from the part of the specification handling the receipt of a message. If the action is \text{Receive} and the queue is nonempty, then remove the data from the queue. Recall that each message has an associated sequence number. If the sequence number of the message is outside of the receiver window, then drop the message. If the sequence number is less than the pointer to the last acknowledged message (\( la \)), then the message has already been acknowledged. This may occur when the acknowledgement is lost in transit or the acknowledgment does not arrive before the timeout. In the event of such an occurrence, the protocol simply sends back
an acknowledgment for that sequence number. If the sequence number is within the window, then place the message in the rcvd buffer. A number of operations are also performed to properly update the pointers as well as to move a block of data to the Output queue. Details of these operations can be found in [35]. If none of the previous cases apply, then set nop is true. The PVS specification follows:

\[
\text{next}(s:\text{WinReceiver}, a:\text{WinReceiverAction}) : \text{WinReceiver} =
\]

\[
\begin{align*}
\text{IF } a &= \text{Receive} \text{ THEN} \\
&\quad \% \text{Check that a data message has arrived.} \\
&\quad \text{IF } \neg \text{isempty_fifo}(s'\text{link_output}) \land \\
&\quad \quad \text{isDataFrame(topof}(s'\text{link_output})) \text{ THEN} \\
&\quad \quad \text{LET } \text{idx} = \text{index(topof}(s'\text{link_output})), \text{ priv} = s'\text{privates} \\
&\quad \quad \text{IN} \\
&\quad \quad \% \text{If the sequence num on the message is greater than the window,} \\
&\quad \quad \% \text{then ignore the message and drop if from link layer.} \\
&\quad \quad \text{IF } \text{idx} \geq \text{priv'nd} + \text{rw} \text{ OR} \\
&\quad \quad \quad \text{priv'la} \leq \text{idx} \land \text{idx} < \text{priv'nd} \text{ THEN} \\
&\quad \quad \quad s \text{ WITH [} \\
&\quad \quad \quad \quad \text{output} := s'\text{output}, \\
&\quad \quad \quad \quad \text{link_input} := s'\text{link_input}, \\
&\quad \quad \quad \quad \text{link_output} := \text{dequeue}(s'\text{link_output}), \\
&\quad \quad \quad \quad \text{privates} := \text{priv}, \\
&\quad \quad \quad \quad \text{nop} := s'\text{nop} \\
&\quad \quad \quad \] \\
&\quad \% \text{In the next case the message has already been acknowledged} \\
&\quad \% (seq num LT la) then send another ack. \\
&\quad \text{ELSIF } \text{idx} < \text{priv'la} \text{ THEN} \\
&\quad \quad \text{LET } \text{frame} = \text{AckFrame(idx,idx)} \text{ IN} \\
&\quad \quad s \text{ WITH [} \\
&\quad \quad \quad \text{output} := s'\text{output}, \\
&\quad \quad \quad \text{link_output} := \text{dequeue}(s'\text{link_output}), \\
&\quad \quad \quad \text{link_input} := \text{enqueue(frame}, s'\text{link_input}), \\
&\quad \quad \quad \text{privates} := \text{priv}, \\
&\quad \quad \quad \text{nop} := s'\text{nop} \\
&\quad \] \\
&\quad \% \text{The sequence number of the message is within} \\
&\quad \% \text{the window so put data in receiver window and set mask to true.} \\
&\quad \text{ELSE } \% \text{In this case, s'private'nd LE idx LT s'private'nd + rw AND} \\
&\quad \quad \text{LET } z = \text{topof}(s'\text{link_output}) \text{ IN} \\
&\quad \quad \text{LET } \text{data} = z'\text{data},
\end{align*}
\]
The PVS fragment given above is representative of specifications of both the sender and receiver processes. Most of the space is devoted to record updates, queue operations moving data from one layer to another, as well as calls to functions such as `first_false` and `deliver`.

The generic fifo queue structure was discussed in Section 3.3.8. The type possesses operators:

- `topof` that returns the top of the queue.
- `dequeue` that removes the top of the queue.
- `enqueue` that adds an element to the back of the queue.
- `isempty` that returns true if the queue is empty.

The translation from the PVS specification of the stack operations is straightforward.

Recall that data being sent is of type `WindowFrame` as discussed in Section 3.3.6. Data messages have both an index field holding the sequence number and the data field with appropriate projection operators `index` and `data` defined. The `isDataFrame` operator returns true if a message is a data message as opposed to an acknowledgment message. The translation of these operators is straightforward.

As discussed in Section 3.3.3, records are translated into Java classes and destructive record updates in PVS are translated into calls to an `update` function in Java that operates on that class. Hence each update of the state `s` in the specification of the form
s with [ .... ]
is translated into a call to WinReceiver’s update operation.
The translated JAVA code is given as follows:

```java
public WinReceiver next(final WinReceiver s,
                      final int a) {
    if (a == 1) {
        if (!fifo_windowframetheory isempty_fifo(s.link_output) &&
            windowframetheory isDataFrame(fifo_windowframetheory topof(s.link_output))) {
            final int idx =
            windowframetheory index(fifo_windowframetheory topof(s.link_output));
            final WinReceiverPrivate priv = s.privates;
            if (idx >= priv.nd+rw || priv.la <= idx && idx < priv.nd) {
                return s.update(s.link_input,
                                fifo_windowframetheory dequeue(s.link_output), s.nop, s.output, priv);
            } else {
                if (idx < priv.la) {
                    final WindowFrameTheory.WindowFrame frame =
                    windowframetheory AckFrame(idx,idx);
                    return s.update(fifo_windowframetheory enqueue(frame, s.link_input),
                                    fifo_windowframetheory dequeue(s.link_output),
                                    s.nop, s.output, priv);
                } else {
                    final WindowFrameTheory.WindowFrame z =
                    fifo_windowframetheory topof(s.link_output);
                    final int data = z.data;
                    final WindowTheory.Window nrcvd =
                    windowtheory.put_data(priv.rcvd.data, idx–priv.nd);
                    final int ff = windowtheory first_false(nrcvd);
                    final WinReceiverPrivate upd_priv =
                    priv.update(priv.la, Math.max(priv.lr, idx+1),
                               priv.nd+ff, windowtheory.slide(nrcvd, ff));
                    return s.update(s.link_input,
                                    fifo_windowframetheory dequeue(s.link_output),
                                    s.nop, deliver(s.output, nrcvd, ff), upd_priv);
                }
            }
        } else {
            return s.update(s.link_input, s.link_output, true, s.output, s.privates);
        }
    }
}
```

The PVS specification of the sender is 144 lines long and the Java code produced by our translator is 153 lines long. The PVS specification of the receiver is 161 lines long and the
Java code produced by the translator is 178 lines long.

The sender and receiver specifications can be interpreted as separate processes, but translating the PVS specification into Java code that creates a pair of threads operating on shared data structures is beyond the current scope of the PVSWhy translator. Instead, we craft this code by hand. At the top layer we constructed a function GDP that initializes a structure holding the two queues that compose the ether and the queue feeding data to the sender and the queue that the receiver places data in. The primary task performed by GDP is to start the sender and receiver threads passing in the shared structure to both. We also crafted the Java functions ThreadedSlidingWinSender and ThreadedSlidingWinReceiver to drive the respective threads. That is, they sit in a loop executing the corresponding state machine’s next function using different values for the action.

At present, we do not automatically generate assertions for the Java code. Yet we have proven a number of invariants of the PVS specification and consequently, it would be possible to apply software verification tools to check that these invariants hold in the Java code. Some of the core invariants are as follows

- \( na \leq ns \leq na + sw \)
- \( 0 \leq la \leq nd \)
- \( nd \leq lr \leq nd + rw \).

An interesting test of one’s verification system would be if it could handle the “global” invariants that formalize the relation between a sender and receiver. For instance:

- The counter of received messages is less than or equal to the counter of sent messages:

  \[ lr \leq ns \]

- The counter of delivered messages is less than or equal to the counter of sent messages:

  \[ nd \leq ns \]

The invariants can also aide in the generation of test cases when the translation is used as a reference implementation.

5 RELATED WORK

Two major fields of computer science come together in generating code from formal specifications: theorem proving and compiler construction.

Within the theorem proving community all the major theorem provers have some form of code generation to a functional language from their specification language. The theorem prover Isabelle/HOL even provides two code generators. There is the original generation from higher order logic to ML, described by Berghofer and Nipkow [5]. A second translator, developed by Haftmann [20], targets multiple languages. Unlike our generator, however, these languages are all functional programming languages like Haskell, OCaml and SML.
ACL2’s [27] specification language is a subset of Common Lisp. The theorem prover Coq [6] has its generator [30] that extracts lambda terms and translates them in either Haskell or OCaml. As mentioned before, PVS [38] provides a code generator for Lisp. A PVS translation into the functional programming language Clean is in its prototype stage [24]. Using semantic attachments or analog mechanisms to tie executable code and logical statements together has been studied by Ray in ACL [42], and by Rushby et al [14] and Muñoz [34] in PVS.

From within the compiler construction community, work has been done on source to source translators from functional languages to imperative ones: A source code translator between Lisp and Java has been constructed by Leitao [29]. However, not all language constructs of Lisp are supported. Another translator from ML to Java was proposed by Koser et al in [28]. Instead of Java, Ada has also been used as a target language by Tolmach [48].

6 CONCLUSION AND FUTURE WORK

Integrating formal methods into the software engineering process requires tools that provide support without unnecessarily constraining the design and implementation choices. We present a tool designed to generate annotated code from declarative PVS specifications for multiple functional and imperative target languages. The key features of our tool are:

- Independently verifiable code: The generated code is accompanied by annotations that allow for proof obligation generation.
- The generated code is readable and it allows for integration with existing code.
- The generated code is reasonably efficient, due to the nature of the translation from an executable subset, as well as by using destructive update optimization techniques. Since we are using an intermediate language, further optimizations such as tail recursion elimination can be easily added.

A preliminary prototype of the tool has been implemented to generate Java code. This prototype was demonstrated on a sliding-window protocol.

The code generator presented in this paper is only a proof of concept. Many features have to be improved to be really useful in a large scale software engineering process. For example, currently, only a subset of the specification language of PVS can be translated. One feature that limits the applicability of our code generation process is that many models are only partially executable. In particular, formal models of protocols typically use a relational specification style to describe functional behaviors. These models cannot directly be translated into an executable program. Being able to generate code for these models, by providing syntactic restrictions on their specification, is one of our next goals. For this we need to add support for guarded non-determinism.

In the spirit of proof carrying code [36], another venue of progress would be to extend the Why logic and the extraction mechanism so that annotated programs carry with them a reference to the correctness lemmas in the original specification and enough information for discharging the proof obligations from these lemmas. Thus, eliminating most of the burden of mechanically proving the correctness of the generated code.

We recognize that all the individual elements are not really novel by themselves, as demonstrated by the related work. However, we believe that tying them all together in a
complete package and targeting both functional and imperative languages, is an important, and needed, contribution in the area of code generation from proof assistants.

REFERENCES


