View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

NASA/TM–2009-215943

# From Verified Models to Verifiable Code

*Leonard Lensink*
*Radboud University Nijmegen, The Netherlands*

*César A. Muñoz*
*NASA Langley Research Center, Hampton, Virginia*

*Alwyn E. Goodloe*
*National Institute of Aerospace, Hampton, Virginia*

October 2009

**NASA STI Program . . . in Profile**

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at 443-757-5803

- Phone the NASA STI Help Desk at 443-757-5802

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM–2009-215943

# From Verified Models to Verifiable Code

*Leonard Lensink*
*Radboud University Nijmegen, The Netherlands*

*César A. Muñoz*
*NASA Langley Research Center, Hampton, Virginia*

*Alwyn E. Goodloe*
*National Institute of Aerospace, Hampton, Virginia*

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

October 2009

## Acknowledgments

This work was supported by the National Aeronautics and Space Administration at Langley Research Center under NASA's Exploration Technology Development Program (ETDP), Cooperative Agreement NCC-1-02043, and NASA's Integrated Vehicle Health Management (IVHM) project, Cooperative Agreement NNX08AE37A. The authors would like to thank all the reviewers and, specially, Kurt Woodham from NASA, for their valuable comments on earlier versions of this paper.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

## Abstract

Declarative specifications of digital systems often contain parts that can be automatically translated into executable code. Automated code generation may reduce or eliminate the kinds of errors typically introduced through manual code writing. For this approach to be effective, the generated code should be reasonably efficient and, more importantly, verifiable. This paper presents 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 programming languages. Several case studies are presented to illustrate the tool's functionality. The generated code can be analyzed by software verification tools such as verification condition generators, static analyzers, and software model-checkers to increase the confidence that the generated code is correct.


## 1 Introduction

Complex critical systems [38] such as fault-tolerant avionics and air traffic management systems pose particular challenges due to the potential loss of life that could incur from a failure. Safety guarantees for such systems require explicit evidence that the systems are sound with respect to their safety requirements – a task that is typically beyond the scope of traditional testing techniques or model checking. NASA has been applying heavyweight formal methods to such problems for many years. Specifically, NASA Langley Research Center has used the Prototype Verification System (PVS) [31] to model and mechanically prove that these models satisfy safety properties such correctness [27, 28], and completeness properties such as validity and agreement [25, 35]. Scientists at the National Institute of Aerospace (NIA) and NASA LaRC have also invested eight years of research in designing formally verified algorithms for air traffic conflict detection and resolution [6, 8, 9]. In total, the number of PVS theorems proved at LaRC now numbers in the thousands. Once these properties are verified, the models are implemented using traditional imperative programming languages. An improvement to this scenario would be to automatically generate code and formal assertions from the formally verified models. This way, the possibility that errors are introduced during the coding phase is reduced. This will also enable the use of verification condition generators or other software verification tools to check that the generated code correctly implements the specified algorithm.

Generally, two different techniques are employed for generating code from formal specifications. The first technique uses the Curry-Howard isomorphism to extract programs from constructive proofs [24, 34]. The second technique translates the original specification into code assuming that the specification is restricted to a pseudo-executable subset of the specification language [1, 18, 40, 43]. The latter technique is particularly appealing for generating code from specifications written in declarative languages, such as PVS and ACL2 [19], since these languages encourage writing specifications in a style that is in large part functional, which supports relatively easy transition to an executable form. Indeed, specifications in ACL2 can

be compiled and run natively in a Common Lisp environment. Furthermore, in the absence of constructive proofs, the second technique is usually the only option.

In this paper, a prototype generator of annotated code for declarative specifications written in PVS is presented. The prototype is electronically available from `http://research.nianet.org/fm-at-nia/PERICO/`. Currently, the prototype generates Java code with assertions written in JML [5]. The code generator uses an intermediate language which supports translation to other target programming languages.

The remainder of the paper is structured as follows. Section 2 presents an overview of the translation process. Section 3 describes the more challenging aspects of the translation. Section 4 gives an overview of several case studies where the tool has been applied to generate Java code from PVS specifications. Finally, the last section discusses related work and concludes.

## 2 From PVS to Java and Back Again

The input to the code generator is a declarative specification written in PVS, a higher order logic specification language and theorem prover. Since this work aims at a wide range of applications, the target language is not fixed. Indeed, the tool first generates code in Why, an intermediary language for program verification [12]. The current prototype generates Java annotated code from the Why code. In the future, the generator may be extended to support other functional and imperative programming languages.

Another benefit of an intermediate language is that transformations and analysis that are independent from the target language, such as tail-recursion elimination and shared memory analysis, can be directly applied to the intermediate code. This allows optimizations to be performed only once that otherwise might have to be performed for every target language. Furthermore, the code generator exports Why code into XML. This insulates the developer of the translation to a specific target language from the internals of the generator or having to write a custom parser. XML parsers are readily available for most modern programming languages.

In order to increase confidence in the generated code, the generator annotates the code with logical assertions such as pre-conditions, 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 such as Krakatoa [11]. Krakatoa generates proof obligations for several theorem provers, including PVS. 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 the prototype. The rest of this section gives a short overview of PVS and Why.

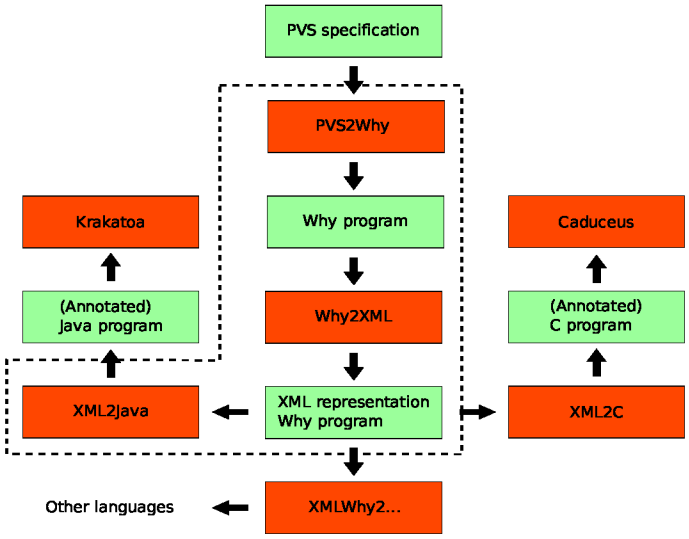


Figure 1. Multi-target generation of verifiable code

## 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 [32].

The specification language is based on classical higher order logic, augmented with a sophisticated type system that uses predicate subtypes and dependent types. It also has the capability to define algebraic data types. All functions that are defined in the specification language must be total, i.e., functions must be defined for all the input values. 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 make type checking in general undecidable. The theorem prover generates type correctness conditions (TCC's) for the undecidable parts of the type checking process. 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 or 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 Filliâtre et al. [10, 13]. The Why tool generates proof obligations for different kinds of ex-

isting 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 Milner's ML programming language 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 [10].

The Why tool is used as the back-end of verification condition generators. Indeed, the same team that develops Why also develops the tools Krakatoa and Caduceus, which are front-ends for Java and C verification condition generators, respectively.

# 3 Code Generation

For the most part, the translation from a declarative PVS specification into the Why language is straightforward. Each language construct in the functional 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.

In order to ease the translation, the Why language has been extended with several features such records, tuples, and a simple notion of modules. Although records and tuples could be defined in the logic part of the language, they have been added as syntactic sugar and treated similarly to arrays. Modules provide a naming scope for a set of Why declarations. They correspond directly to the parameterized theories in PVS and allow for modularity in the generated programs. A more general notion of module that includes the notion of interface is currently being added to the Why core language [42].

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

```
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. That function can be translated into Why as a *proposition*:

```
predicate is_square(x:real,y:real) = (y*y=x)
```

which can be used in logical assertions, or it can be translated as a *program*:

```
let is_square(x:real,y:real):bool
  y*y=x;
```

which can be used in other programs. The same distinction applies to general functions that 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, the appropriate Why code has to be generated based on how PVS expressions are being used in the formal model. Section 3.1 discusses which parts of PVS specifications are used to generate logical assertions.

One of the main concerns in generating code from a declarative specification is the efficiency of the resulting program. Purely functional programs may be inefficient. 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, 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`. Because Why destructive updates are more efficient that PVS overriding, it is particularly tempting to translate the PVS overriding feature with Why statements such as `A[0] := 10`. Here, the index `0` of the array `A` is set to `10`. Why destructive updates are more efficient than PVS overriding. However, as it will be seen in Section 3.2, a careful analysis has to be performed to guarantee the correctness of this translation.

The last step is the translation of Why programs into the target language. Except for higher-order functions, all the constructs in the Why language can be directly translated into most modern programming languages. Also needing definition is a process for integrating the generated code with existing code. The solution to these two processes is highly dependent on the target language. As an example, Section 3.3 discusses how higher order functions and code integration are supported in the generation of Java code.

## 3.1 Assertions

The predicate subtyping capability of PVS allows for the precise specification of functions equivalent to pre-conditions and post-conditions in traditional Hoare logic-based specification languages [37]. For instance, the square root function in PVS can declared:

$$\texttt{sqrt}(\texttt{x:real} \mid \texttt{x} \geq 0) : \{\texttt{y:real} \mid \texttt{y} \geq 0 \wedge \texttt{x} = \texttt{y*y}\}$$

This declaration states that **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-conditions 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 on the use of programs in logical statements.

The Why language does not have the means to specify proofs, but allows for the specification of axioms used by automated theorem provers to discharge the proof obligations. All lemmas and TCC's are translated into axioms in the Why logic.

## 3.2 Destructive Updates

The PVS ground evaluator includes a highly efficient code generator that translates PVS expressions into Lisp [39]. 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 it is not possible to alias the variable being overridden, the destructive version is chosen; if there is a risk of aliasing, the safe version that makes a copy is used. The alias analysis performed by the PVS ground evaluator is made difficult or even impossible by nested application and higher-order functions. Consequently, the approach applied by the evaluator is conservative in that it applies the safe version if the alias evaluation is infeasible.

Take for instance the following PVS function that negates each element of an array of 1000 elements:

```
Arr : TYPE = ARRAY[below(1000) → 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
```

The destructive update of the array `A` can be done safely because the update of element `i-1` is done after the value of the element has been read and there is no reference to `i-1` afterward. If the update is done non-destructively, an array copy would have to be performed 1000 times with significant performance results. As it is not always possible to use the destructively updating variant of the function `negate`, the translator generates a destructive and a non-destructive version of each function that updates a variable. For instance, if the function `negate` is used in a situation where the negated array is referenced later, i.e. `foo(negate(A),A)`, the version of `negate` function that employs the non destructive update is used.

The translation from PVS to Why uses a similar mechanism as the PVS ground evaluator. However, due to the aliasing exclusion mechanism built into the type system of Why, it is impossible to translate functions that perform array updates into a non-destructive and destructive version of the same function. Instead, a destructive variant of every function is generated. If the alias analysis determines that a particular variable cannot be destructively updated, a deep copy of this variable is created before performing any destructive update. Hence, computations are safely performed without destroying the initial object and avoiding the possible introduction of aliasing. The function call `foo(negate(A),A)` is translated into `B = A.copy; foo(negate(B),A)`.

## 3.3 Java

Several techniques have been proposed to provide support for higher-order functions in Java. In this work, a special class `Lambda` that encodes closures as objects is used. Then, every function in Why is translated into a Java function and a static object of type `Lambda` that encodes the curryfied closure of the function. Overloading allows for the use of the same name for the Java function and for its closure.

Inheritance and abstract classes are used to enable the integration of the generated code with existing code. This is particularly useful when a given function is uninterpreted in the original specification. Take for example the case of the square root function in PVS. Since a constructive version of this function is not available, this function and the class where it appears are declared as abstract in Java. This means that the program that invokes the generated code must provide a concrete **sqrt** function in order to execute the code. Since the pre-conditions and post-conditions of **sqrt** are still generated, Krakatoa, or any other verification condition generator for Java, should generate proof obligations that guarantee the provided function satisfies the specification of the uninterpreted one. For example, the **sqrt** function is translated into the following JML annotated Java code:

```
//@ requires x >= 0
//@ ensures \result >= 0 AND x = \result * \result
public abstract float sqrt(final float x);
```

### 3.4 Example

In order to illustrate some of the more specific properties of the translation from PVS to Java, the translation of a small part of the consensus protocol example given in Section 4.3 is shown.

Several actions are defined in the specification of this protocol that can happen in the model. These are captured using an abstract datatype in PVS. In this datatype, both constructors and recognizers are defined. PVS creates functions for these definitions that can be used in any expression and also creates accessor functions that will return the parameter used in construction.

```
Actions : DATATYPE
BEGIN
  Good              : Good?
  Garbled           : Garbled?
  Sym(frame:Frame)  : Sym?
  Asym(frame:Frame): Asym?
END Actions
```

All specifications and lemmas are defined inside what PVS calls a *theory*. Theories are analogous to modules in a programming language and translate into classes in an object-oriented language. The datatype `Actions` is defined inside the theory `ReceiveAction` and translated into a Java class.

The abstract datatype `Frame` is defined in a different theory parameterized by a type variable `Data`. The notion of parameterized theories is captured in the translation by generic class declarations.

Each of the different actions becomes a subclass that extends the class `Actions` and has a constructor that takes the generic class `Frame` as an argument.

```
public class ReceiveAction <Data> {
  public class Actions { public Actions() {}}
  public class Sym extends Actions {
    FrameTh<Data>.Frame frame;
```

```
  public Sym(FrameTh<Data>.Frame frame) {
    this.frame = frame; } }
```

The acccessors and recognizers that are implicitly defined in the PVS theory are explicitly added to the Java code. Note that it is impossible to use the `instanceof` operator: in order to make generic types work on existing JVM architectures, all the generic type information is destroyed when compiling to byte code.

```
public FrameTh<Data>.Frame SymAccessor(Sym sym) {
  return sym.frame; }
public boolean SymRecognizer(Actions actions) {
  return actions.getClass().getName().equals("Sym"); }
```

The higher order use of defined functions is facilitated by a special `Lambda` class. This generic abstract class demands that an application function is supplied for each instance.

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

For all defined functions in PVS, a higher order version is generated that satisfies the requirements of the `Lambda` class.

```
  public Lambda<Actions,Boolean> SymRecognizer =
    new Lambda<Actions,Boolean>() {
    public Boolean apply(final Actions actions) {
      return SymRecognizer(actions); } }; }
```

All functions are translated in a currified version. This way it is possible to translate all higher order uses of functions, including partial application into working Java programs.

# 4 Case Studies

In this section, three case studies are briefly presented where the PVS to JAVA tool has been used by NASA to create executable prototypes from a PVS specification. The first case study is of an algorithm intended for use in air-traffic management, the second is a sliding-window protocol, and the third is a Byzantine consensus algorithm.

## 4.1 KB3D

KB3D [8] is a pair-wise conflict detection and resolution (CD&R) algorithm developed at the former research institute ICASE at NASA Langley Research Center. The input to KB3D is the position and velocity vectors of two aircraft. KB3D distinguishes the host aircraft as the *ownship* and the traffic aircraft as the *intruder*. The output is a list of resolution maneuvers for the ownship. The maneuvers computed by KB3D are new velocity vectors that involve the modification of a single parameter of the ownships original flight path: vertical speed, track, or ground speed. KB3D is not computationally intensive and suitable for distributed airborne deployment.

KB3D is characterized by the following features:

- Distributed: Each aircraft solves its own conflicts with respect to traffic aircraft.
- Three dimensional: It proposes horizontal and vertical resolutions.
- State-based: It solves conflicts based only on the state information of each aircraft, i.e., current position and velocity vector.
- Tactical: It uses a short lookahead time, typically 5 minutes or less.
- Geometric: It finds analytical solutions, in a Cartesian coordinate system, assuming linear trajectory projections of current aircraft states.

The mathematical properties of KB3D have been extensively studied and formalized in PVS.

### Code generation for KB3D

For this experiment, only the conflict detection algorithm of KB3D, namely `cd3d`, is considered. This algorithm is a relatively simple, but critical, component of KB3D. The Java code generated for `cd3d` serves as a reference implementation that can be used to reduce the chance of inadvertently introducing errors. Take for instance part of the original PVS specification:

```
cd3d(sx,sy,sz,vx,vy,vz,D,H,T) : bool =
 IF  vx=0 ∧ vy=0 ∧ sq(sx)+sq(sy) < sq(D) THEN
            (abs(sz) < H) OR
       (vz*vz < 0 ∧ -H < sign(vz)*T*vz + sz)
 ELSE ....
 ENDIF
```

The first manually coded version of `cd3d`, which was programmed before the code generation for PVS was developed, contained an error in the implementation. At some point in the algorithm, a "≤" symbol was used instead of the original "<" symbol of the specification:

```
public boolean cd3d(
        double sx, double sy, double sz,
        double vx, double vy, double vz,
        double D, double H, double T) {
      if (vx==0 && vy==0 && Util.sq(sx)+Util.sq(sy) < Util.sq(D)) {
        return
         Util.abs(sz) < H ||
         (vz*sz <= 0 && -H <= Util.sign(vz)*T*vz + sz);
```

This did not make the implementation of `cd3d` incorrect, but it did mean that the algorithm was no longer complete. Of course, the mechanically translated reference implementation does not contain this error.

## 4.2 Sliding-Window Protocol

AirSTAR [2] is a dynamically scaled experimental aircraft designed and built by NASA's Langley Research Center (LaRC) for use as a testbed for research on software health management and flight control. The AirSTAR team commissioned two of the authors to study a small protocol that provides a guarantee of eventual message delivery, but would be simpler, and more verifiable than say User Datagram Protocol (UDP)/Transmission Control Protocol (TCP), which are considered to be too complex to be used in AirSTAR. This procotol is called the Guaranteed Delivery Protocol (GDP). Following the standard solution to this problem, GDP [29] is designed as a sliding-window protocol with block acknowledgment [14,15]. A formal specification of the protocol was written in PVS.

In addition to the formal specification of the protocol and its correctness proofs, a reference implementation was developed.

### Code generation for AirSTAR

The main challenge in generating code for the AirSTAR model was the use of parameterized (generic) theories, abstract datatypes, and higher order functions in the PVS specification.

As shown in Section 3.4, the use of Java generics makes it possible to refrain from having to generate Java classes for each particular instance of the parameterized theory.

Higher-order functions were mainly used in the `Ether` part of the model, where bags are used to represent the physical layer holding or duplicating frames during transit. In PVS, bags are represented by functions that return the number of the same frame present in the physical layer.

```
bag: TYPE = [LinkFrame → nat]
emptybag : bag = (λ (t:LinkFrame): 0)
insert(x:LinkFrame,b:bag) : bag =
 (λ (t:LinkFrame): IF x = t THEN b(t) + 1 ELSE b(t) ENDIF)
```

Functions over a finite type are represented as arrays. However, `LinkFrame` is not a finite type, therefore a higher order representation is generated using the same class `Lambda` as defined in Section 3.4.

```
public int emptybag(final LinkInterfaceTheory.LinkFrame t) {
 return 0; }
public Lambda<LinkInterfaceTheory.LinkFrame,Integer>
 insert(final LinkInterfaceTheory.LinkFrame x,
        final Lambda<LinkInterfaceTheory.LinkFrame,Integer> b) {
    return new Lambda<LinkInterfaceTheory.LinkFrame,Integer>() {
      public Integer curry(final LinkInterfaceTheory.LinkFrame t) {
        return (x == t ? b.curry(t)+1 : b.curry(t)); } }; }
```

## 4.3 Fault-Tolerant Consensus

Verification and validation (V&V) of distributed fault-tolerant systems is a continuing challenge for safety-critical systems. A combination of technologies to provide

V&V support for these distributed fault-tolerant algorithms needs to be explored. The aims of this case study is to investigate ways to aide test engineers by generating test cases that exercise the system under fault models used during formal analysis. The process involves applying the PVS to Java translator to a PVS model and then using Symbolic Java PathFinder (SJPF) [33] to generate test cases from the Java code. SJPF has the capacity to generate test cases using symbolic execution techniques, in particular: lazy initialization, to create symbolic data that acts as input instead of concrete data. Symbolic execution produces a structure that encodes constraints on the data fields and an execution path condition. Off-the-shelf constraint solvers are then used to generate the concrete structures that are input to the program. As an example, this process has been applied to a simple variant of the oral-messages Byzantine consensus protocol [22]. A PVS model of the protocol is then analyzed to show validity and agreement. Applying the translator, a Java program that can then be analyzed by SJPF was obtained.

### Code generation for Fault-Tolerant Consensus

The PVS specification of the Byzantine protocol employs most of the language constructs used in the sliding window protocol: theories parameterized by types, abstract datatypes, and higher order functions. A complicating factor was the use of arrays of generic abstract datatypes such as in the definition of the receiver:

```
NICReceiver  : Type  = [#
   from_nic : ARRAY[below(maxsize) → fifo[Frame]],
   wires    : ARRAY[below(maxsize) →fifo[Frame]],
   pc       : Stage,
   nop      : bool #]
```

In Java, it is not possible to declare arrays of a generic type. Instead, an Array class is used, in which a function is defined that can construct arrays if the size is known and an initialization function is passed. The constructed `ArrayList` is subsequently cast to a regular array.

```
public class Array<T> {
  public T[] new_array(int size,Lambda<Integer,T> lambda) {
    ArrayList<T> array = new ArrayList<T>(size);
    for (int i=0;i<size;i++)
    array.set(i,lambda.curry(i));
    return (T[]) array.toArray(); } }
```

The `new_array` function is used to generate the generic lists that are needed in the generated code.

## 4.4 Evaluation

The three case studies show that it is feasible to generate Java code from PVS models for small to medium sized models. Using the models helped to improve the quality of the generated code and to extend the Why language with several constructs that are either syntactic sugar, such as record types, or completely new additions, such as the notion of modules.

The models roughly double in size when translated into Java, as is shown in the following table of model sizes and translation times.

| | PVS | Java | time |
|---|---|---|---|
| Conflict Detection | 122 lines | 282 lines | 0.4s |
| F.T. Consensus | 252 lines | 657 lines | 1.0s |
| Sliding Window | 1094 lines | 2162 lines | 1.4s |

Although the translator supports all executable language constructs that PVS provides, some minor changes had to be made to the original models to support translation. The changes all have to do with clashing name spaces, non-translatable characters in identifiers and the fact that the translation requires all fields of a record to be updated. All these issues will be resolved in the final release. Furthermore, the generation of pre-conditions and post-conditions is still work in progress.

The case studies presented in this section should be viewed in terms of "proof of concept." Although several features are still missing, the case studies demonstrate the potential for integrating heavy-weight formal methods tools into the software development cycle.

# 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. The original generation from higher order logic to ML described by Berghofer and Nipkow [3], and a second translator, developed by Haftmann [16], which targets multiple languages. Unlike the generator presented here, these languages are all functional programming languages like Haskell, OCaml and SML. ACL2's [20] specification language *is* a subset of Common Lisp. The theorem prover Coq [4] has a generator [24] that extracts lambda terms and translates them in either Haskell or OCaml. As mentioned before, PVS [31] provides a code generator for Lisp. A PVS translation into the functional programming language Clean is in its prototype stage [17]. Using semantic attachments or analog mechanisms to tie executable code and logical statements together has been studied by Ray in ACL [36], and by Rushby et al [7] and Muñoz [26] in PVS.

Integrating formal methods into the software engineering process has been the main goal of the B-method [1]: a collection of mathematically based techniques for the specification, design and implementation of software components. The primary departure for the method presented here is that PVS as a specification language allows for higher order functional specification and is a more powerful theorem prover than those that come with the B-tool suite. The added expressiveness of the specification language allows for code generation to functional languages, unlike the B-method where only C or ADA code can be generated.

A similar approach is taken with the Vienna Development Method (VDM) [18]. This also is a collection of formal methods and tools that aim at using mathematical techniques in the software development process. It does support higher order functions and can generate Java as well as C++ code. However, their code generator uses a standard library of VDM concepts instead of translating more directly into the target language. Both VDM and the B-method do not annotate the generated code, which makes it harder to check whether the generated code is indeed correct.

From within the compiler construction community, work has been done on source-to-source translators from functional languages to imperative languages: a source code translator between Lisp and Java has been constructed by Leitao [23]; however not all language constructs of Lisp are supported. Another translator from ML to Java was proposed by Koser et al in [21]. Instead of Java, Ada has also been used as a target language by Tolmach [41].

# 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. This paper presents a tool designed to generate annotated code from declarative PVS specifications for multiple functional and imperative target languages. The key features of the 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 an intermediate language is used, further optimizations such as tail recursion elimination can be easily added.

The code generator presented in this paper is only a proof of concept. Many features have to be improved to be deployable 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 the 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 the next goals. For this, support for guarded non-determinism is needed.

In the spirit of proof carrying code [30], 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.

This will eliminate most of the burden of mechanically proving the correctness of the generated code.

It is recognized that all the individual elements are not original by themselves, as demonstrated by the related work. However, it is believed 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


1. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

2. Roger Bailey, Robert Hostetler, Kevin Barnes, Celeste Belcastro, and Christine Belcastro. Experimental validation subscale aircraft ground facilities and integrated test capability. In *Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005*, San Francisco, California, 2005.

3. Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *LNCS*, pages 24–40. Springer, 2002.

4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

5. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joseph Kiniry, Gary Leavens, Rustan Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

6. Ricky Butler and César Muñoz. A formal framework for the analysis of algorithms that recover from loss of separation. Technical Memorandum NASA/TM-2008-215356, NASA, Langley Research Center, Hampton VA 23681-2199, USA, 2008.

7. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and David Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001. Available at `http://www.csl.sri.com/users/rushby/abstracts/attachments`.

8. Gilles Dowek, Alfons Geser, and César Muñoz. Tactical conflict detection and resolution in a 3-D airspace. In *Proceedings of the 4th USA/Europe Air Traffic Management R&DSeminar, ATM 2001*, Santa Fe, New Mexico, 2001. A long version appears as report NASA/CR-2001-210853 ICASE Report No. 2001-7.

9. Gilles Dowek and César Muñoz. Conflict detection and resolution for 1,2,...,N aircraft. In *Proceedings of the 7th AIAA Aviation, Technology, Integration, and Operations Conference, AIAA-2007-7737*, Belfast, Northern Ireland, 2007.

10. Jean-Christophe Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

11. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

12. Jean-Christophe Fillitre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Universit Paris Sud, March 2003.

13. Jean-Christophe Fillitre and Claude March. Multi-prover verification of C programs. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer Verlag, November 2004.

14. Mohamed Gouda. *Elements of Network Protocols.* Wiley-Interscience, 1998.

15. Mohamed Gouda and Nicholas Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

16. Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number 364/07, 08 2007.

17. Bart Jacobs, Sjaak Smetsers, and Ronny Wichers Schreur. Code-carrying theories. *Formal Asp. Comput.*, 19(2):191–203, 2007.

18. Cliff Jones. *Systematic Software Development using VDM (second edition).* Prentice Hall, 1990.

19. Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

20. Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

21. Justin Koser, Haakon Larsen, and Jeffrey Vaughan. SML2Java: a source to source translator. In *In Proceedings of DP-Cool, PLI03, Uppsala,Sweden*, 2003.

22. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982.

23. Antonio Menezes Leitao. Migration of Common Lisp programs to the Java platform -the Linj approach. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 243–251, Washington, DC, USA, 2007. IEEE Computer Society.

24. Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

25. Paul Miner, Alfons Geser, Lee Pike, and Jeffrey Maddalon. A unified fault-tolerant protocol. In Y. Lakhnech and S. Yovine, editors, *FORMATS/FTRTFT*, Lecture Notes in Computer Science 3253, pages 167–182, 2004.

26. César Muñoz. Rapid prototyping in PVS. Contrator Report NIA 2003-03, NASA/CR-2003-212418, NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, May 2003.

27. César Muñoz, Víctor Carreño, and Gilles Dowek. Formal analysis of the operational concept for the Small Aircraft Transportation System. In *Rigorous Engineering of Fault-Tolerant Systems*, volume 4157 of *Lecture Notes in Computer Science*, pages 306–325, 2006.

28. César Muñoz, Víctor Carreño, Gilles Dowek, and Ricky Butler. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer*, 4(3):371–380, 2003.

29. César Muñoz and Alwyn Goodloe. Design and verification of a distibuted communication protocol. Technical Report NASA/CR-2009-215703, National Aeronautics and Space Administration, 2008.

30. George Necula. Proof-Carrying Code. In *Proceedings of 24th Symposium on Principles of Programming Languages (POPL97)*, pages 106–119. ACM Press, 1997.

31. Sam Owre, John Rushby, and Natarajan 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, jun 1992. Springer-Verlag.

32. Sam Owre, John Rushby, Natarajan Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, oct 1998. Springer-Verlag.

33. Corina Pasareanu, Peter Mehlitz, David Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Soft-

ware. In *International Symposium on Software Testing and Analysis*, pages 15–26. ACM Press, 2008.

34. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *J. Symb. Comput*, 15(5/6):607–640, 1993.

35. Lee Pike, Jeffrey Maddalon, Paul Miner, and Alfons Geser. Abstractions for fault-tolerant distributed system verification. In *Theorem Proving in Higher-Order Logics*, Lecture Notes in Computer Science 3223, pages 257–270. Springer-Verlag, 2004.

36. Sandip Ray. Attaching Efficient Executability to Partial Functions in ACL2. In M. Kaufmann and J S. Moore, editors, *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*, Austin, TX, November 2004.

37. John Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, sep 1998.

38. John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, jan 1993.

39. Natarajan Shankar. Efficiently executing PVS. Technical report, Menlo Park, CA, 1999.

40. Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In *11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 01)*, volume 2372 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 2002.

41. Andrew Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

42. Wendi Urribarrí. A module system for Why. Manuscript, Personal Communication, 2008.

43. John Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
**PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| 01-10-2009 | Technical Memorandum | |

**4. TITLE AND SUBTITLE**
From Verified Models to Verifiable Code

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Lensink, Leonard; Munoz, Cesar A.; Goodloe, Alwyn E.

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**
645846.02.07.07.15.03

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
NASA Langley Research Center
Hampton, VA 23681-2199

**8. PERFORMING ORGANIZATION REPORT NUMBER**
L-19766

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
National Aeronautics and Space Administration
Washington, DC 20546-0001

**10. SPONSOR/MONITOR'S ACRONYM(S)**
NASA

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**
NASA/TM-2009-215943

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified - Unlimited
Subject Category 61
Availability: NASA CASI (443) 757-5802

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

Declarative specifications of digital systems often contain parts that can be automatically translated into executable code. Automated code generation may reduce or eliminate the kinds of errors typically introduced through manual code writing. For this approach to be effective, the generated code should be reasonably efficient and, more importantly, verifiable. This paper presents 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 programming languages. Several case studies are presented to illustrate the tool's functionality. The generated code can be analyzed by software verification tools such as verification condition generators, static analyzers, and software model-checkers to increase the confidence that the generated code is correct.

**15. SUBJECT TERMS**
Code extraction; Functional specification; Prototype Verification System; Software verification; Formal methods

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | STI Help Desk (email: help@sti.nasa.gov) |
| U | U | U | UU | 24 | 19b. TELEPHONE NUMBER *(Include area code)* (443) 757-5802 |