Generation of certifiably correct programs from formal models

Alexei Iliasov
Newcastle University
Newcastle upon Tyne, UK
alexei.iliasov@ncl.ac.uk

***Abstract*—Application of formal notations and verifications techniques helps to deliver systems that are free from engineering defects. A code generator is an essential tool for formal development of real-world systems; it transforms models into runnable software quickly, consistently and reproducibly. Commonly, a code generator is a program constructed informally and producing an output that is not formally traced to an input. Industrial standards to the development of safety-critical systems, such as IEC 61508, require a justification for any tool used in a development: extensive prior experience or a formal certification. An extensive experience is often not an option as there are very few sufficiently mature modelling toolsets. The certification of a code generator is a major effort increasing costs and development time. We propose an approach where a modeller places no trust whatsoever in the code generation stage but rather obtains software that is certifiable without any further effort. The essence of the approach is in the transformation of a formal model into runnable software that is demonstratively correct in respect to a given set of verification criteria, coming from a requirements document. A Hoare logic is used to embedd correctness criteria into the resultant program; the approach supports design-by-contract annotations to allow developer to mix formal and informal parts with a fair degree of rigour.**

***Keywords*-code generation; formal modelling; certifying compiler; Event-B**

I. Introduction

A formal development is a journey starting with a requirements document and ending with the construction of a product. In between, there is a modelling stage based on a formal notation and some form of formal verification. When applied properly, formal verification gives an ultimate form of evidence that the developed design meets all the criteria of the respective specification - a formal rendering of a requirements document. In this paper we focus on the stage of a development where a design, represented in a formal, mathematical notation, is converted into a product; in the context of this work such a product is taken to be a piece of software represented as a text in a high-level programming language.

A piece of software is obtained from a model using an automated tool - a code generator. Manual code construction has too many downsides to consider it a part of a safety critical development process: it is slow, error prone, and, as a development process, has no verification or certification potential. An automated code generator works much like a compiler and makes the transition from a model to a program an instantaneous and reproducible process. A code generation tool is an essential ingredient of a formal development process if one hopes to achieve certification via the route of formal verification. Such a code generator must construct programs that preserve the properties proven at the level of the formal model. Realising such a tool is far from trivial.

Typically, a code generator is a large and fairly intricate program that mechanically transforms a formal design into software. Demonstrating the correctness of a code generator has proven challenging due to the scale and complexity of code generation logic and the semantical mismatch between source and target notations. It seems natural to require that a code generator is up to a certain standard. Thus, applying a code generator to a safety-critical design one should expect software that is error-free[1]. This mode of thinking has been portrayed in a number of industrial standards. For instance, DO-178B, an avionics industry software standard, regards a code generator as a development tool and requires that it is certified to the same integrity level as the system which code is produced by the code generator. IEC 61508 standards family, widely applied in areas such as nuclear power, factory automation and automotive sector, require that a code generator is developed at least to the same safety integrity level as the product to which development it is applied. For the highest safety integrity level (SIL 4) - the most fitting level for developments applying formal verification - this means that a code generator itself must be developed to SIL 4 standards. One reason why formally developing a code generator is rarely cost efficient is that code generation is a domain specific activity. In contrast to programming language compilers that use the same transformation step for each and every input program, a code generator must cover a sizeable semantical gap, that is, provide an interpretation to an abstract formal design. Such an interpretation is necessarily tied to the purpose, the pragmatics, of an input design, expressed in a requirements document, and this essentially requires that each problem domain uses its own

[1]Assuming an error is an action in violation of the original design there is no logically consistent transformation that would convert a perfect design into imperfect software. This does not rule out errors not captured in the formal design, errors caused by misinterpretation of requirements and errors to problems in the requirements document.

code generation tool. Including the construction of a code generator in a development process means higher costs and a longer development cycle. These two factors might tip the balance away from the use of formal modelling. In addition, certification requirements vary considerably between the certification bodies of differing nations and industries.

II. Code Generator: Certified vs. Certifying

An industry standard to development tools such as IEC 61508 demands an evidence that any tool applied in a development process is fit for purpose: good enough for the criticality level of the developed software. Such an evidence may come in the form of the past experience or could be inferred from the circumstantial evidence on the development processes and methods applied in the creation of the tool. The most direct route, however, is the process of tool certification undertaken by some recognized third-party authority. IEC 61508 does not itself define the procedure of certification but there are a number of relevant national and international standards, e.g., DO-178B[1]. Ideally, a code generator for a formal modelling notation would be certified from the evidence of a formal proof of its correctness. Unfortunately, fully capturing and validating a code generator in discrete logic is extremely hard without relying on another code generator that would cover the gap between an abstract description of transformation logic and runnable code. This chicken and egg problem is rooted in the fact that while formal notations excel in the representation of abstract ideas they do not generally work so well for the description of concrete implementation details. The problem may be resolved if a modelling notation embeds a form of a programming language (one example is the B0 language extending the Classical B notation to add programming constructs [2]) and the associated verification technology scales to the level necessary to realise a realistic code generator.

Once a code generator is granted a certificate, the certificate is tied to specific application domain and target platform. The definition of a target platform includes hardware and any software that would run together with the generated system software. Such tight coupling means the re-certification is likely in subsequent projects if they target newer or differing platforms.

In the absence of a viable certification route for a code generator one could simply make use of an uncertified tool and subject the resultant code to all the validation activities necessary for the certification of a developed system. This makes a formal development far less attractive as the already costly formal verification activity does not reduce the overall certification costs.

An alternative is to change the subject of certification from a code generation tool to the output of such tool. That is, do not attempt to certify the tool but rather certify the programs produced by the tool. At a first glance it seems we are back in the situation of an uncertified code generator. There is a way, however, to make the certification of programs constructed by a code generator very cheap if not completely automatic. This is possible with a technique called proof carrying code - executable code augmented with the proof of its correctness [3]. A tool producing such code is often called a certifying compiler [4]. Provided the correctness properties are sufficiently strong and adequate in respect to the original requirements, the proof constructed by a certifying compiler is an ultimate evidence that the program is error-free (but see the preceding footnote).

Such an evidence may be assessed in an objective and mechanised manner using a proof checker - a tool that automatically checks whether a given proof agrees with a certain verification goal. Proof checker is a relatively simple tool for which certification is feasible. Although we are not aware of the existence of any such tool, there is no lack of well-regarded and mature theorem provers. A theorem prover is by definition also a proof checker. In its interactive mode it accepts commands from a user and checks whether they are a part of an implemented logic. In a proof checking mode, it processes a complete proof script validating each individual inference rule of a proof.

Assuming there is a satisfactory proof checker, there remains the questions of where the program properties come from and how corresponding proofs are constructed. The work on proof carrying code focuses on the verification of program texts. Such verification activity may provide assurance for a rather small class of low-level properties such as buffer overflow/underflow, access to uninitialised memory and similar problems that are reflected in the semantics of a source notation but may not be detected easily using conventional compilation-time checks. A certifying compilers defines a formal semantics for an input notation and generates verification conditions that check that the semantical constraints are respected by an input program. The conditions, in the form of theorems, typically in the form of a first-order logic, are delegated to an automated theorem prover. If all the conditions are discharged, the compiler is able to provide a correctness certificate together with the output program (usually, object code). No assurance is given if there is a problem in the input program or the prover is not powerful enough to conduct an autonomous proof. One can also expresses additional verification goals but in practice there are severe limitations on the kind of properties that may be efficiently verified.

In the scope of a formal development it is natural to consider a formal design as the source of verification conditions. The role of a certifying code generator is then to transfer the properties of an input model to the output program; to achieve this, a code generator must be able to construct proofs demonstrating the satisfaction of such properties in the output program. Since the original formal design already contains proofs for such properties it is tempting to attempt

mechanical translation of these proofs into the context of the resultant program.

III. Introduction to Event-B

The basis of our discussion is a formalism called Event-B[5]. It belongs to a family of state-based modelling languages that represent a design as a state (a vector of variables) and state transformations (computations updating variables). In general, a design in Event-B is abstract: it relies on data types and state transformations that are not directly realisable. This permits terse models abstracting away from insignificant details and enables one to capture various phenomena of a system with a varying degree of detail. Crucially, each statement about the effect of a certain computation is supported by a formal proof. In Event-B, one is able to make statements about safety (this incorporates the property of functional correctness) and progress. Safety properties ensure that a system never arrives at a state that is deemed unsafe (i.e., a shaft door is never open when a lift cab is on a different floor). Progress properties ensure that a system is able to achieve its operational goals (i.e., a lift cab eventually arrives).

Being a general-purpose formalism, Event-B does not attempt to fit any specific application domain. It has found applications in modelling of hardware, validation of high-level use case scenarios, verification of business process logics and even as a friendly notation for a mathematician looking for a support from machine provers. In the development of software, such expressive power is helpful in capturing environmental phenomena not related to software, e.g., behaviour of a human operator or laws of physics governing the kind of inputs a system receives.

An Event-B development starts with the creation of a very abstract specification. The cornerstone of the Event-B method is the stepwise development that facilitates a gradual design of a system implementation through a number of correctness-preserving *refinement* steps. The general form of an Event-B model (or *machine*) is shown in Figure 1. Such a model encapsulates a local state (program variables) and provides operations on the state. The actions (called *events*) are defined by a list of new local variables (parameters) vl, a state predicate g called *event guard*, and a next-state relation S called *substitution* (see the **EVENTS** section in Figure 1).

The **INVARIANT** clause contains the properties of the system (expressed as state predicates) that should be preserved during system execution. These define *safe states* of a system. In order for a model to be consistent, invariant preservation should be formally demonstrated. Data types, constants and relevant axioms are defined in a separate component called *context*.

Model correctness is demonstrated by generating and discharging a collection of proof obligations. There are proof obligation demonstrating model consistency, such the preservation of the invariant by the events, and the refinement

MACHINE M
  SEES Context
  VARIABLES v
  INVARIANT $I(c, s, v)$
  INITIALISATION ...
  EVENTS
    E_1 = **any** vl **where**
      $g(c, s, vl, v)$
    then
      $S(c, s, vl, v, v')$
    end
    ...
END

Figure 1. Event-B model structure.

link to another Event-B model. A collection of automated theorems attempts to discharge generated proof obligations; typically only 3%-5% of proofs require user intervention. Putting it as a requirement that an enabled event produces a new state v' satisfying the model invariant, the model *consistency* condition states that whenever an event on an initialisation action is attempted, there exists a suitable new state v' such that the model invariant is maintained - $I(v')$. More details on Event-B and its methodology may be found in [5] and on the Event-B community website[6].

If a model possesses rich control flow properties (e.g., a computational algorithm) the control flow aspect of a model is defined in a separate view called the flow of a model [7]. The flow aspect could require demonstrating further constraints but offers a guarantee that the prescribed event ordering is present in the model. In this work we apply the flow aspect to obtain structured programs - programs that use concepts like sequential composition, choice and loop. Not always a flow aspect needs to be defined and some systems (i.e., embedded control systems for platforms like TinyOS) are data-driven even at the implementation stage.

IV. Certifying Code Generator for Event-B

To evaluate the idea, we have implemented a proof of concept tool integrated with the Event-B development toolkit - Rodin Platform[6]. The tool is available for evaluation together with a set of sample problems [8]. Currently, the output notation is either a pseudo-code or a subset of Java with JML annotations [9]. The main difference between pseudo-code and Java is the use of Java expression language in place of Event-B mathematical notation. At the moment, we do consider the problem of translating proofs originally constructed using set-theoretic notation into equivalent proofs for the target programming language notation. It is expected that an automatic procedure implementing such a translation would yield valid proof scripts on the basis of proofs done for the source, modelling notation.

Not all variables of a model must contribute to computations of a generated program. This is mainly due to a methodological requirement that developed software is modelled together with the properties of its prospective

operational environment (derived from the environmental assumptions of a requirements document). At the model level, one does not formally distinguish between these two parts. However, to construct an executable code it is necessary to draw a boundary between a system and its environment. In our approach, a developer is asked to indicate whether a variable is included in an output program and if not, what is the role of this variable. We have defined the following two classes of non-computational variables.

Modelling variables: A variable that is used solely for the purpose of demonstrating certain helper properties may be defined as a modelling variable. If the properties dependent on such variable are not required by the requirements document the variable may be projected away from all actions, guards and invariants. The projection procedure replaces a free variable in formula with a bound variable. For instance, projecting out x from $F(x, v)$ gives $\exists z \cdot z \in \mathrm{type}(x) \land F(z, v)$; here $\mathrm{type}(x)$ is an expression evaluating to the type of variable x. To project a variable from an action, an action is first translated into a predicate form. Thus, projecting out x from $a := a + x$ yields $\exists z \cdot z \in \mathrm{type}(x) \land a' = a + z$ or, using the Event-B event notation, **any** z **where** $z \in \mathrm{type}(x)$ **then** $a := a + z$ **end**. An action assigning to a model variable completely disappears in an output program. The projection rule is also applied to verification conditions. A mistakenly declared modelling variable should make correctness conditions too weak to satisfy system requirements. A special case of a modelling variable is an auxiliary variable: a variable that is only ever used in event guards and actions updating the same or other auxiliary variables. Auxiliary variables are projected cleanly: they do not leave an existential quantifier in formulae.

Environment variables: The behaviour of an environment is not implementable but it plays an important role: it defines the set of conditions under which the developed software is guaranteed to function correctly. If the promise of the environment behaviour is broken there is formally no obligation for a program to continue to function in accordance to the specification. In Event-B, environmental assumptions are used to strengthen safety invariants. In a program, these appear as correctness conditions formulated on environment variables. The behaviour of an environment part is not included into a model but is translated into a set of rely conditions. These conditions formally characterise a valid operational environment and could assists in integration testing. Often the environment of a program is another, informally developed program. In this case, a tool like VeriFast could help to establish the satisfaction of rely conditions.

A. Hoare Logic

As is common to PCC approaches, our program verification technique is based on a variation of the Hoare's proof system. Most of the inspiration comes from [10]. The main departure from the classical presentation is the usage of sets in the roles of pre- and postconditions and, correspondingly, relations in the role of commands. This allows us to replace the original syntactical substitution rules and the assignment axioms with a more flexible notion of a state transition more directly supporting non-deterministic substitutions. Our axiomatic framework is the ZF set theory.

Assume that Ω is a system state. In a triple $\{P\}f\{Q\}$, $P \subseteq \Omega$ is a pre-condition, $Q \subseteq \Omega$ is a post-condition and $f \subseteq \Omega \times \Omega$ is a next-state relation. To prove that statement $\{P\}f\{Q\}$ is correct for an elementary command (relation) f one has to show that $f[P] \subseteq Q$. Relying on the correspondence between sets and predicates, the latter may be expressed in a predicate form: $R_P(v) \land R_f(v, v') \Rightarrow R_Q(v')$ where $P = \{v \mid R_P(v)\}, Q = \{v \mid R_Q(v)\}, f = \{v \mapsto v' \mid R_f(v, v')\}$ and $\Omega = \{v \mid \top\}$. In a similar fashion, we can switch between set theoretic and predicate forms for other verification conditions. Some of the inference rules of the logic are given below.

$$\frac{\{P\}f\{\top\}}{\{P\}f\{f[P]\}}\ \textbf{act}$$

$$\frac{\{P\}f\{Q\},\quad \{R\}g\{S\},\quad Q \subseteq R}{\{P\}f;g\{S\}}\ \textbf{comp}$$

$$\frac{\{P \cap C\}f\{Q\},\quad \{R \setminus C\}g\{S\},\quad T \subseteq P \cap R,\quad Q \cup S \subseteq U}{\{T\}\ \textbf{if}\ C\ \textbf{then}\ f\ \textbf{else}\ g\ \textbf{end}\ \{U\}}\ \textbf{cnd}$$

$$\frac{\{C \cap I\}f\{I\},\quad \exists V \cdot V \subseteq C \cap I \land f[V] \subset V}{\{C \cap I\}\ \textbf{while}\ C\ \textbf{do}\ f\ \textbf{end}\ \{I \setminus C\}}\ \textbf{loop}$$

B. Translation Templates

We illustrate the basic ideas behind the code generator by showing how some generic pieces of model may be transformed into runnable code. The simplest case is the translation of a deterministic event, like the following one

$$\mathsf{e} \quad = \quad \textbf{when}\ x < k\ \textbf{then}\ x := x + 2\ \textbf{end}$$

The translation is a single Hoare triple embedding an assignment command. The guard becomes a pre-condition; a post-condititon is computed from the before-after predicate of the source event.

$$\{x < k\}\ x := x + 2\ \{x < k \land x' = x + 2\}$$

When there is more than one substitution in a source event the output is some suitable serialisation of substitutions. If a substitution is non-deterministic the output program implements an assignment with a help of a procedure call. The body of such procedure is initially empty but its declaration is decorated with pre- and post-conditions derived from the substitution. For such a procedure the correctness certificate is not a proof derived from a formal model but rather the record of a validation effort by a design-by-contract verification tool, i.e., VCC or JML. It is an open question how one can convincingly combine verification evidence originating from two differing verification approaches. As an illustration consider the following event:

$$\mathsf{e} \quad = \quad \textbf{when}\ y < k\ \textbf{then}\ x := x + 2 \parallel y :| y' < x\ \textbf{end}$$

The translation is as follows.

$\{x < k\}\ x := x + 2\ \{x' = x + 2\};$
$\{x < k \wedge x' = x + 2\}\ ?\ \{y' < x \wedge x < k \wedge x' = x + 2\}$

Here the question mark is a magic assignment command. In an output programming language text it is replaced by function or a method call, as explained above. If an event makes use of a parameter, -

e = **any** z **where** $x < z$ **then** $x := z - x$ **end**

- then it is necessary to introduce a local variable to simulate the role of the parameter. In this example we have

$\{\exists u \cdot x < u\}$ **def** z **for**
$\{\top\}\ ?\ \{x < z'\}$
$\{x < z'\}\ x := z - x\ \{x < z' \wedge x' = z' - x\}$
end $\{\exists u \cdot x < u \wedge x' = u - x\}$

The first assigment is a procedure call initialising local variable z. Depending upon the meaning of z, it could a simple computation, a prompt to a user to supply a value for z or reading a message from a communication buffer. Whatever such a procedure calls does, it must satisfy the post-condition $x < z'$. A local variable may not be referenced by name outside of the scope of its declaration. Wherever it is semantically relevant, as it is the case in the initial pre-condition and the final post-condition of the example aboce, a local variable appears in a projected form.

Since primed variables are defined only in the modelling part, it is forbidden to differentiate between the values of primed and unprimed variables in assignments of a program. Because of this, some events cannot be serialised without the help of a local variable. In our approach, such a variable is always responsible for the value of an unprimed variable. Consider event

e = **begin** $x := y \parallel y := x$ **end**

and its translation.

$\{\top\}$
def xp **for**
$\{\top\}\ xp := x\ \{xp' = x\};$
$\{xp' = x\}\ x := y\ \{x' = y\};$
$\{xp' = x \wedge x' = y\}\ y := xp\ \{xp' = x \wedge x' = y \wedge y' = xp\}$
end
$\{x' = y \wedge y' = x\}$

A nice property of this code is that the auxiliary variable, necessary for implementability, is semantically irrelevant. By a case of the one-point rule, it is eliminated from the final post-condition.

A program is obtained by translating each event containing normal (non-model, non-environment) variables. The overall program structure is that of a loop containing a number of **if** statements, one for each translated event. The condition of a statement is the associated event guard and the body is the translated event body. An alternative translation, targetting event-driven control systems, assumes that a loop is a part of the environment, a guard is an event subscription condition and translated event actions define an event handler. However, even in the latter case, it is often necessary to compose events to define event handlers.

If the flow aspect is present, the control flow information is used to generate a structured program. It is fairly straightforward to traverse a flow graph and infer the skeleton of program containing the usual structuring constructs. In a general case, not every flow graph has an equivalent representation in a structured program (assuming **goto** statements are not allowed). One problematic pattern is passing control from within of a loop body into the middle of another loop and returning back but at a slightly earlier point. This may lead to an infinite unfolding of a procedure constructing structured loops.

C. Example: GCD

In this section we study a small synthetic example concerned with the construction of a function computing the greatest common devisor (GCD) of two numbers. Imagine we are confronted with the following requirement.

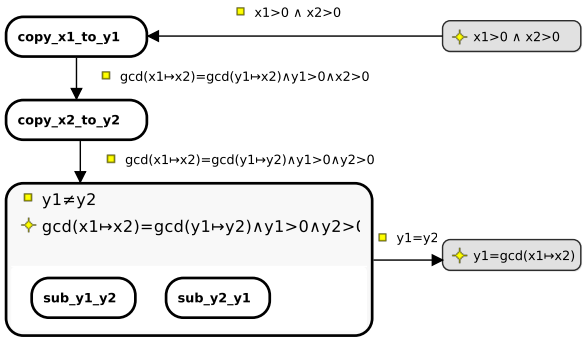
REQ179: given two positive numbers, compute the GCD of the numbers where GCD is defined as

$P0 : \gcd \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
$P1 : \forall a, b \cdot a, b \in \mathbb{N} \wedge a > b \Rightarrow \gcd(a, b) = \gcd(a - b, b)$
$P2 : \forall a, b \cdot a, b \in \mathbb{N} \wedge b > a \Rightarrow \gcd(a, b) = \gcd(a, b - a)$
$P3 : \forall a \cdot a \in \mathbb{N} \Rightarrow \gcd(a, a) = a$

There are many ways to construct an Event-B model addressing the requirement. We use an approach where most of the verification activity is placed in the flow aspect. This uncharacteristic for larger Event-B developments where the flow aspect would have a fairly low-key contribution to the verification activity. In the example, the Event-B model defines simple computation steps and states a safety invariant that is nothing more than a typing condition.

MACHINE gcd_flow
VARIABLES $x1, x2, y1, y2$
INVARIANT $x1 \in \mathbb{N} \wedge x2 \in \mathbb{N} \wedge y1 \in \mathbb{N} \wedge y2 \in \mathbb{N}$
EVENTS
copy1 = **begin** $y1 := x1$ **end**
copy2 = **begin** $y2 := x2$ **end**
sub1 = **when** $y1 \neq y2$ **then** $y1 := y1 - y2$ **end**
sub2 = **when** $y1 \neq y2$ **then** $y2 := y2 - y1$ **end**
END

The flow aspect of the model makes a provable statement that, starting with some two positive numbers, a certain sequence of event executions results in the computation of the GCD of the numbers.



In the flow diagram above, rounded rectangles are events, arrows represent the possibility of passing control from one event to another. Grey rectangles are assertions and are used to formulate theorems related to the flow aspect of a model. The large rounded box is a structured a loop. It contains

two events as a loop body (semantically, a choice of the two events) and is annotated with a loop condition and invariant. A structured loop is a syntactic sugar that makes a diagram visually more compact, especially when a loop invariant is used.

Applying the code generator produces the following program.

```
1   {x1 > 0 ∧ x2 > 0}
2   y1 := x1   [act, set theory]
3   {α ∧ y1 = x2 ∧ gcd(x1, x2) = gcd(y1, x2)}
4   y2 := x2   [act, set theory]
5   {α ∧ y1 = x2 ∧ y2 = x2 ∧ I}
6   while y1 ≠ y2 do
7       inv : I
8       {y1 ≠ y2 ∧ I}
9       if y1 > y2 then
10          {y1 > y2 ∧ I}
11          y1 = y1 − y2   [act, P1]
12          {I}
13      else
14          {y1 < y2 ∧ I}
15          y2 = y2 − y1   [act, P2]
16          {I}
17      end   [cond]
18      {I}
19  end   [loop, P3, set theory]
20  {y1 = gcd(x1, x2)}
```

where $\alpha = x1 > 0 \wedge x2 > 0$, $\beta = y1 > 0 \wedge y2 > 0$ and $I = \alpha \wedge \beta \wedge \gcd(x1, x2) = \gcd(y1, y2)$. The shown program is only a part of the generated output. The second part is an a proof library formed from a subset of proofs done for the source model and containing proofs for all the inference rules $[\ldots]$ and the invariant properties. Ideally, such proofs would be a derivation from the original model proofs and account for the difference between the mathematical language of Event-B and expression language of the target programming language. At the moment, we limit ourselves to problems where these two syntactically coincide although we are aware of possible semantical mismatches (i.e., implementable integers vs. infinite range integers).

V. Conclusion

The growing importance of software certification could signify an increase in the adoption of formal techniques. It is necessary to understand how certification changes the practice of software development and how this may be used to enhance the position of formal modelling. Code generation seems to be one of the weak links of a formal development toolchain. An informally constructed code generator almost nullifies the gains of the preceding verification stage. A possible answer, as argued in this paper, is to apply the proof carrying code technique to build a certifying code generator.

There is a number of challenging questions for which we do not dare to claim a prompt and successful resolution. Automatic translations of proofs between model and program notations is perhaps the most demanding point.

In this discussion we have ignored the issue of correctness and certification of a compiler for a high-level programming language. Clearly, our approach depends on the acceptance of any such compilation tool in a development process. We do no expect a sudden appearance of certified compilers but rather see the current work as a prepation for an approach to emitting proof-carrying machine code. There a number of promising results in this direction most notably the work on verification of JVM in HOL [11] and a certifying compiler for Java [12].

Due to space constraints, we were not able to discuss a larger example. We invite an interested reader to try the tool and available sample models[8].

Acknowledgments

This work is supported by FP7 ICT DEPLOY Project and the EPSRC/UK TrAmS platform grant.

References

[1] RTCA Special Committee 167, "Software Considerations in Airborne Systems and Equipment Certification," RTCA, Inc., Tech. Rep., 1992.

[2] J.-R. Abrial, *The B-Book*. Cambridge University Press, 1996.

[3] G. C. Necula, "Proof-carrying code," in *Proceedings of the Symposium on Principles of Programming Languages*, N. D. Jones, Ed. Paris, France: ACM Press, 1997, pp. 106–119.

[4] G. C. Necula and P. Lee, "The design and implementation of a certifying compiler," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98, 1998, pp. 333–344.

[5] J.-R. Abrial, *Modelling in Event-B*. Cambridge University Press, 2010.

[6] Event-B, "Community web site," 2011, http://event-b.org/.

[7] A. Iliasov, "Use case scenarios as verification conditions: Event-B/Flow approach," in *Proceedings of 3rd International Workshop on Software Engineering for Resilient Systems*, Septembre 2011.

[8] Rodin platform plug-in, "B2H5," 2011, available at http://iliasov.org/b2h5/.

[9] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, pp. 212–232, June 2005.

[10] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[11] G. Klein and T. Nipkow, "Verified bytecode verifiers," *Theoretical Computing Science*, vol. 298, pp. 583–626, April 2003.

[12] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline, "A certifying compiler for Java," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, ser. PLDI '00, 2000, pp. 95–107.