On the Requirements of High-Integrity Code Generation[*]

Michael W. Whalen Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
{*whalen, heimdahl*}*@cs.umn.edu*

Abstract

Although formal requirements specifications can provide a complete and consistent description of a safety-critical software system, designing and developing production quality code from high-level specifications can be a time-consuming and error-prone process. Automated translation, or code generation, of the specification to production code can alleviate many of the problems associated with design and implementation. However, current approaches have been unsuitable for safety-critical environments because they employ complex and/or ad-hoc methods for translation.

In this paper, we discuss the issues involved in automatic code generation for high-assurance systems and define a set of requirements that code generators for this domain must satisfy. These requirements cover the formality of the translation, the quality of the code generator, and the properties of the generated code.

1 Introduction

Software plays an increasingly important role in safety-critical systems as computers take over crucial functionality. A system is safety-critical if the incorrect operation of the software could lead to loss of life, substantial material or environmental damage, or large monetary losses.

One of the main sources of flaws in safety-critical systems is inadequate software system specifications. Specifications written entirely in natural language (e.g., English) can be inadequate because natural language is inherently ambiguous and difficult to analyze. Formal specification languages are designed to precisely and concisely state the requirements for a software system. When used in tandem with explanatory text, formal specification languages can be used to create complete and unambiguous specifications for a software system. Nevertheless, even if a formal requirements effort produces a correct specification, designing and developing production quality code from the specification can be a time-consuming and error-prone process.

Automatically generating the production code from a specification alleviates many of the problems in system development. If done correctly, automated translation guarantees that the behavior of the production code is correct with respect to the formal specification; the specification as well as the program can be viewed as mathematical objects so such a translation is clearly feasible. There are commercial systems that are based on this idea and translate a formal or semi-formal specification to executable code, for instance, Statemate from I-Logix [6], the Rose tools from Rational Corporation, and SCADE from Verilog [4].

There are, however, several reasons to distrust any code generated from a formal or semi-formal specification: (1) the specification language and/or target language may lack formal semantics, (2) the translation may not be formally defined, and (3) the translation tool may be incorrectly implemented. Consequently, the full benefits of code generation are not realized in safety-critical systems. However, we believe that appropriately adopted, code generation can and should be used to satisfy customer quality, regulatory, legal, and ethical requirements imposed on safety-critical applications. To achieve an acceptable level of confidence that the generated code is correct, a wide range of mechanisms must be used in concert.

In this paper we explore the requirements of code generation for safety-critical systems in detail. Section 2 provides an overview of code generation from formal specifications. Section 3 describes the level of formality necessary to reason about the translation process. Section 4 describes requirements for developing the implementation of the translation. Section 5 describes readability and traceability constraints on the generated code. In Section 6, we describe an ongoing project to create an environment satisfying our stringent requirements. Finally, we briefly survey

[*]This work has been partially supported by NSF grants CCR-9624324 and CCR-9615088, and University of Minnesota Grant in Aid of Research.

several code generators for different classes of formal languages and discuss their applicability for code generation for safety critical systems.

2 Code Generation Overview

The goal of a high-assurance development effort is to produce a well-understood, well-documented software system that works correctly in its target environment with minimal cost and effort. Provably-correct code generation allows a development team to greatly reduce the time spent in the design and implementation stages of the development process, and eliminates all errors potentially introduced throughout these stages. It also facilitates a development process where any changes to the system are made directly to the specification, not the implementation code, so that the specification and the implementation are always kept consistent. Thus, the goal of code generation from high-level formal requirements models is to increase productivity and quality by directly deriving production code from the formal model.

To take full advantage of code generation from formal specifications it is imperative that the formal specification is correct. If the specification cannot be guaranteed to be correct, the generated code must undergo the same extensive V&V required for manually created code; the only saving would be the elimination of the actual coding stage. On the other hand, if we can validate and verify the specification itself, we may be able to greatly reduce the V&V resources that need to be expended on the generated code; this is where the significant cost savings can be realized. Thus, *any correctness preserving code generation is preconditioned on a correct specification.*

Unfortunately, having a fully formal and correct requirements model does not guarantee the success of a development approach based on automated code generation; the translation and translation tool must also be correct. In addition, the translation must be *acceptable* to the customer, regulatory agencies, and any legal interests. Regulatory standards related to safety critical software development have traditionally taken a cautious (if not skeptical) view of code generation and their concerns must be satisfied before this approach will achieve widespread use.

To provide the level of confidence in the code generation approach needed for its use on safety-critical software in a regulated and litigious society, we must take every precaution possible to demonstrate that the code generation works correctly. We can achieve acceptable confidence if the code generation approach satisfies the following requirements:

- The source and target languages are formally defined.
- The translation between the source and target is formally proven to be meaning-preserving.
- Formal arguments are provided to validate the translator software and/or the generated code.
- The translator software is rigorously tested and treated as high-assurance software.
- The generated code is well-structured, well-documented, and easily traceable to the original specification.

The next three sections discuss these requirements in more detail.

3 Formal Basis

Any high-assurance technique must be grounded in mathematics. If we cannot formally state what specifications and programs mean, then there is no hope of showing that a specification and a program are equivalent. However, if a mathematical description of the source and target language is provided, then we can specify the code generation process as a transformation function, and we can reason about whether the transformation function creates a program with the same meaning as the specification.

The first step is to define the precise meaning of the source and target languages.

Requirement 1
The source and target languages must have formally well-defined syntax and semantics.

All of the high-level specification languages in which we are interested have a formal semantics. Examples include Z [14], SCR [8], and RSML [10]. Target languages present more of a problem. Since most functional and logic languages use recursion and dynamic memory allocation, they are not suitable for safety-critical applications and must be excluded from consideration. In the remaining category, imperative languages, it is difficult to find a language with a manageable semantics. The denotational semantics for SPARK-Ada (a subset of Ada used for safety-critical applications) requires over 500 pages of Z [11]. A similar effort to formalize Modula-2 also required several hundred pages [15]. The likelihood that such voluminous language semantics definitions are correct is small.

The reason that these languages require such enormous specifications is that the languages were defined without thought of full formalization. Thus, to provide rigorous arguments for the correctness of a translation, it is not feasible to use any of the existing general purpose programming languages as a target language. Instead, one must target a much smaller language by removing complicating features such as sequencers (i.e., statements that cause unusual transfer of control, for instance, goto, break, and continue), and pointers, that do not belong in safety-critical applications. Such a simplified language can be powerful enough to implement useful high-assurance systems, yet simple enough to have a fully formalized semantics.

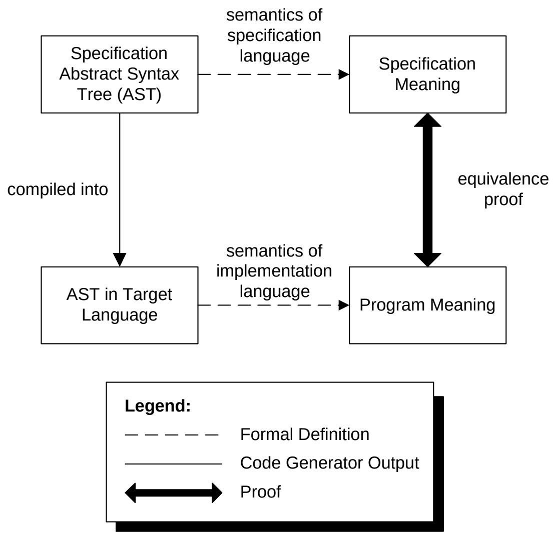

Figure 1. Formal code generation process

Once we have source and target languages with formal semantics, it is possible to create a translation between them. The translation process, as described in Figure 1, describes a provably equivalent representation of the specification in the target language.

Requirement 2
The translation between a specification expressed in a source language and a program expressed in a target language must be formal and proven to maintain the meaning of the specification.

What it means for a translation to *maintain the meaning* of a specification depends on the formalism used to describe the specification, and in the type of program to be translated. For example, a specification language for embedded systems describes the behavior in terms of a transformation function from an initial state and input variable values to a new state and output variable values. In this case, the translation is meaning-preserving if, for any state and input set valid for the specification, the specification and the program generate the same output values and state information. Note that a program can be meaning-preserving to a specification and still have “extra” state information that is used for scratch space, loop counters, etc.

Because even relatively simple languages can have large semantic descriptions, the proofs that the translation between the source and target languages maintains the meaning of the specification become quite large. To simplify the proofs, it is necessary to break them apart into much smaller subproofs. If both the source and target language are specified using denotational semantics (or some variant), it is possible to use structural induction to break the proofs into much more manageable pieces [15] (Figure 1).

To summarize, a code generation approach for high-assurance systems requires fully formal semantics for both the source and target languages. In addition, the translation between a specification expressed in the source language and the implementation captured in the target language must be fully formal and proven to preserve the meaning of the specification.

4 Implementating the Translation

A formal translation mathematically describes how to correctly convert a specification to a target program, but it does not provide us with a useful software artifact. An *implementation* must be created that correctly performs the translation. In the implementation stage, it is not currently possible to use a completely formal approach; it would require a formally verified language, compiler, operating system, and hardware platform. Therefore, we attempt to use rigorous arguments about the translator and/or the generated code in tandem with testing and validation techniques used in safety-critical software.

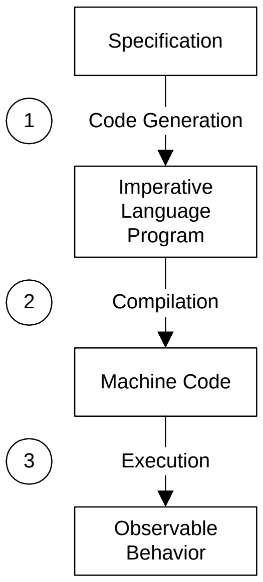

Figure 2. Formal code generation process

The goal is to create an approach that automatically generates machine code based on a formal specification through a process as shown in Figure 2. In this report we are concentrating on the requirements of step 1 of the process, but the compiler (step 2) and the hardware (step 3) must also work correctly for the full approach to be valid; similar correctness requirements should also be placed on these steps.

Requirement 3
Rigorous arguments must be provided to validate the translator and/or the generated code.

There are two techniques to validate code generated by a translator. The first approach is to validate the translator itself. Ideally, the verified design of the code generator would be implemented using a validated compiler in a provably-correct language that provides a high level of abstraction. To the best of our knowledge, there are no languages that match these criteria. The next-best choice is to use well-known and well-understood features in a high-level language with a rigorously validated compiler. Languages such as Ada and Prolog can be used for this purpose. Since current state of the art is unable to extend formal proofs to this endeavor, it is imperative that the implementation is as simple as possible, and is transparently based on the formal translation discussed in the previous section. The simpler and clearer the mapping between the formal semantics and the implementation of the translator, the easier it is to make correctness arguments regarding the translator.

An alternate approach is to create a *proof checker* that will validate a given translation from specification to code. The generated code is annotated with correctness obligations which can be verified against the original specification by another tool. The proof checker tool takes the original specification and the annotated code and determines whether the program correctly implements the specification. In this case, the proof checker, not the translator itself becomes the tool that must be rigorously validated. This approach is most useful when the compiler is very complicated or performs extensive optimizations on the generated code.

Requirement 4
The implementation of the translator must be rigorously tested and treated as high-assurance software.

Because the implementation cannot be entirely formally verified, we must expend any reasonable software development efforts to further increase our confidence in the correctness of the translation. In general, the translator implementation or proof checker should be regarded a high-assurance project itself and must, therefore, follow any development and testing standards defined for high-assurance software, for example, DO-178B [13] or MOD-0055 [12].

Thus, every possible effort should be expended on the assurance of the generated code. This involves attempting to keep the tool implementations as simple and clean as possible, applying any feasible formal proof techniques, using a rigorous development effort, and extensively testing the final product.

5 Target Code Attributes

Although the formal arguments generated above provide a high level of assurance that the software will operate as intended, they are not infallible. Considering the complexity of formal semantics and the volume of proof required to verify the translation and, to a greater extent, the translator, it is possible that errors have been introduced and some of the proofs are wrong.

To provide additional confidence that the generated code is correct, the structure of the code must allow independent means of verification, such as manual inspections and testing. Also, some regulatory agencies are reluctant to accept formal proofs as correctness arguments, so an independent verification of the generated code is required for regulatory approval. Therefore, the code output from the code generator must be in a form that adheres to good software engineering practice and follows the standard styles for the target language. This output should include comments describing the generated functions as well as traceability information to the original specification [17].

Requirement 5
The generated code must be well structured, well documented, and easily traceable to the original specification.

The first reason that the output must be readable and traceable is for manual inspections. Manual inspections of the code is an effective means of verification and may be used for additional assurance or be required for certification. If the code is not clear and easily traceable to the requirements model, inspections are not possible.

Traceable output is also necessary for effective system testing. If an error is discovered in testing, we must be able to trace the error back to the specification to determine where the error was introduced. It is also possible that despite our best efforts, the error was introduced by a fault in the code generator or by the compiler that translates the generated code to assembly language. Thus, in order to correct errors in the specification and to ensure that the code generator and compiler create correct code from the specification, we must be able to understand and trace between the generated code and the requirements.

A special testing issue involves verifying timing constraints. Most specification languages view the software as responding infinitely fast to external events. The specification may include timing constraints, but the code generation approach can rarely (if ever) guarantee that these constraints will be met but the generated program. Even if the run-time of the generated code can be bounded, it is difficult to verify the end-to-end timing constraints of the system because the generated code usually must interact with externally-written code, such as a real-time operating system (RTOS) or hardware drivers. If this is the case, the performance of the code must be validated after it is generated. If the generated code is readable and traceable, it is possible to instrument the code where appropriate to validate the timing behavior of the system.

By providing readable and traceable code, several forms of validation are facilitated: manual inspections, automated analysis of the generated code, and testing. In order to derive the highest level of assurance for a safety-critical system, all of these approaches are necessary, so that a defect in an automated tool or an oversight by an inspector does not prevent finding an error in the system.

6 The Nimbus Approach

As discussed earlier, the utility of code generation is predicated on having a correct specification. The NIMBUS environment is designed to facilitate all stages of the software development process, including specification creation, analysis, and execution of formal specifications written in the Requirements State Machine Language (RSML) [10]. We are attempting to use the guidelines described above to create a code generator to the NIMBUS environment,

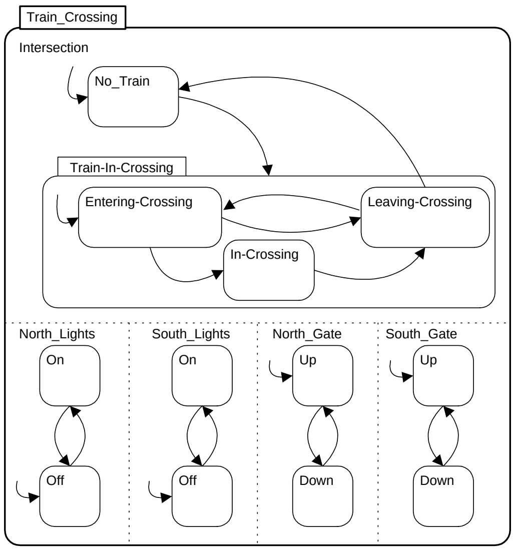

Figure 3. State Machine for Train Crossing

To describe an overview of the code generation process from RSML specifications, we provide a perfunctory, simplified overview of the RSML language. Readers interested in a more complete description of the language are referred to [7]. RSML is based on hierarchical finite state machines and is in many ways similar to David Harel's Statecharts [5]. An example of an RSML specification is shown in Figure 3. The state of an RSML machine (its *MachineState*) is described by:

- The histories of all variables defined in the specification.
- The set of states from the machine that are currently occupied, called the *configuration*.

The behavior of an RSML machine is defined by events and transitions. An RSML specification has a set of predefined events, similar to a finite state automaton. Each transition is triggered by one event. However, a transition may also have a guarding condition, which describes an additional predicate that must be satisfied for the transition to be taken. If a transition is taken, it may change the configuration of the machine, assign values to variables, and cause new events to be generated.

Formally, the behavior of an RSML machine is a relation from *MachineState* to *MachineState*. The relation is created by composing the transition functions within the specification. Each transition can be thought of as a partial function that is defined if its guarding condition is satisfied. These transitions are grouped by their trigger event, then composed in serial ($f \circ g$) and in union ($f \cup g$) to create a relation that describes the behavior of the machine under a particular event. Next, the behavior of the machine under a set of events is the serial composition of its behaviors under each event. The complete behavior of a machine in response to a message is described by the *next state relation F*. If we can show that the behavior of the machine under each event is a total function, then we can show that the behavior of the machine as a whole is a total function [7].

Our approach uses denotational semantics to describe the source and target languages, and structural induction to create the proof of equivalence between the two languages. To show that two specifications are equivalent, we first show that the most basic constructs (constants and variables) are equivalent, then we inductively prove that higher level constructs are equivalent. By using structural induction, we can break the very large proof of specification equivalence into much smaller proofs for each language construct.

Most of the constructs defined in an RSML specification, such as types, variables, and user-defined functions, can be given a standard imperative denotational meaning. Transitions in RSML are defined as partial denotational functions, so they can be composed as described earlier. These transitions create the semantics of the RSML next-state function.

Since the denotational meanings of "real" imperative languages like Ada and C are incredibly verbose, we decided to create denotational semantics for a very simple imperative language specifically designed for use in proofs for code generation. The language, SIMPL (for *S*afety-critical *IMP*erative *L*anguage), contains constructs for variables, constants, functions, procedures, basic and composite (array and record) types. However, it does not support flow of control statements such as breaks, gotos, continues, or early returns from functions, nor does it support pointers. Since SIMPL corresponds to a strict subset of several languages, it is a straightforward process to translate the SIMPL code to a commercial imperative language. Using the SIMPL language also allows a large degree of flexibility for targeting different imperative languages with the code generator.

The basic constructs in RSML such as variables, types, and user-defined functions have a denotational meaning, so it is possible to directly perform equivalence proofs between these entities in the RSML specification and their equivalents in SIMPL. Much of the mapping between the RSML specification and SIMPL is trivial; variables, types, and expressions have a one-to-one correspondence in RSML and SIMPL, and have the same denotation as well. Translating the transitions and the next-state function is more involved, but the compositional nature of the proofs makes each proof relatively straightforward.

The proofs for a given RSML language construct all follow a variant of Figure 4, which is derived from Stepney's research in provably correct compilers research [15]. Given an initial state for the SIMPL program, we translate it to the equivalent RSML state. Then we generate the meaning of the construct both in RSML and in the generated SIMPL code. If the denotation generated is the same for both paths, then the RSML construct and its implementation are proved equivalent.

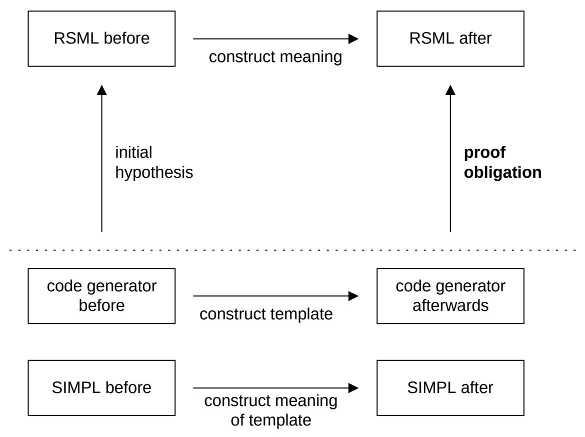

Figure 4. Overview of proof process

Translating the denotational and operational meaning of an RSML specification must be as transparent and straightforward as possible. We plan on using Prolog with extensions for Definite Clause Translation Grammars (DCTGs) to implement the code generator; Prolog provides direct support for parsing specifications, it maps very closely to the formal semantics of the translator, and supports the list and set types required. Furthermore, it has been used successfully for implementing high-integrity compilers [15].

We have also designed the translation to produce human readable output. The most important goal was for the generated code to resemble and to be easily traceable to the original specification. The areas that we have focused our attention are:

Traceability: Each transition in an RSML specification is implemented as a function in the SIMPL program, of the following form:

```
if (predicate) then
   exit source state
   enter destination state
   modify variables if necessary
   add any events generated by
     this transition to event queue
end if
```

Since this notation matches very closely to the input from the RSML file, it is straightforward to examine and trace the generated code.

Readability: If variables that are not present in the RSML file are added to the SIMPL file, they are always well commented and appropriately named.

Extensive Commenting: Every function is described in terms of its type (transition or composition), its location in the RSML machine, and its behavior. To support commenting, we add a dummy statement type to SIMPL for comments.

Performance: Although correctness and traceability were our primary design criteria for code generation, performance is also a concern. With a preliminary version of the code generator in a case study [9], performance was approximately an order of magnitude slower and approximately the same size as hand-coded C. We are researching techniques for improving the performance while satisfying the requirements described in this paper.

With our approach, we have attempted to satisfy the requirements that we have described in this paper. By specifying the denotational semantics of RSML and SIMPL, we satisfy requirement 1. By specifying a formal translation process between RSML and SIMPL, we satisfy requirement 2. By creating an implementation in Prolog that closely matches the formal translation, we satisfy requirement 3. We plan to provide a comprehensive set of specifications to test the implementation to satisfy requirement 4. By adhering to specific goals for the output, we satisfy requirement 5.

7 Related Work

Our guidelines are based on the work of several existing compilers and code generators for various formal specification languages.

7.1 High Integrity Compilers

High integrity compilers are concerned with creating a provably correct mapping between an imperative language and an assembly language for safety critical systems. Many of the requirements for provably correct code generation are originally from research in developing a high-integrity compiler [16, 15]. These systems are verified either by denotational proofs (as outlined in this paper) or through transformational approaches. High integrity compilers have been successfully used on several projects, including nuclear power.

This is a complementary area of research to high integrity code generators. With a high integrity compiler, it is possible to validate the entire compilation process from a high-level specification to machine code. Unfortunately, this field is still in its early stages, and research groups are in the process of scaling up their languages to a point where they would be suitable for large-scale projects.

7.2 Executable specification languages

An executable specification language is a formally well defined, very high-level programming language. Languages such as PAISLey [18] and ASLAN [1] are intended to replace requirements specifications, design specifications, and, in some instances, implementation code. Thus, most executable specification languages are intended to play many roles in the software development process. Executable specification languages have achieved some success and have been applied to industrial size projects. Many languages have elaborate tool-sets and support refinement of a high level specification into more detailed design descriptions or implementation code.

Nevertheless, current executable specifications languages have several drawbacks. Most importantly, the syntax and semantics are, in our opinion, close to traditional programming languages. Therefore, they do not provide the level of abstraction and readability required of a notation if it is going to be usable as a requirements specification language. Furthermore, no currently available language provides support for high level specification of the interfaces governing the interaction between embedded software and the environment. Code generation is not directly supported in PAISLey or ASLAN.

Notable exceptions to the languages discussed above are a collection of state-based notations. Languages such as Statecharts [5], SCR (Software Cost Reduction) [8], and RSML [10], are very-high level and provide excellent support for inspections since they are relatively easy to use and understand.

We have already described the RSML code generation capabilities; Statecharts also supports a code generation facility; however, Statecharts does not have a fully formal semantics, so no formal correctness arguments can be presented for the generated code. SCR currently does not include a code generation tool as part of its environment.

7.3 Synchronous programming languages

Synchronous programming languages such as Esterel [2] and Lustre [3] are programming languages with extensions to support abstract parallelism and control structures based on *events*. These extensions are useful for developing embedded software, especially when a program is supposed to react to several external events in parallel. Esterel uses explicit, program- or environment-generated events to sequence computations, while Lustre is based on control theory and uses a dataflow model and implicit events based on *clocks* which denote time in the external world. Synchronous languages have a formal model, and provide some mechanisms for formal verification. For example, in Lustre, an entity cannot be data-dependent on itself. These languages have been successfully used in large industrial projects, including Airbus flight software. As synchronous languages can be transformed into finite state automata, very efficient code can be generated from the models.

Synchronous programming languages are not, however, designed to be used as specification languages. They are, instead, an improvement on standard imperative languages for implementation of reactive systems. They suffer from the same problems as executable languages when used for specification. Also, although it is possible to create a very efficient automaton for the generated code, the generated automatons bear little resemblance to the original specification, so it is difficult to read the generated code, trace it back to the original specification, and independently verify that the translation was done correctly.

8 Summary and Conclusion

Code generation holds the promise of eliminating much of the time and effort required to implement safety-critical systems, while at the same time eliminating errors introduced in this stage of development. However, without stringent guidelines on the translation, the implementation of the translator, and the structure of the output, this promise will not be realized because the generated code cannot be trusted.

In this paper, we have provided a minimum set of requirements for creating a code generator that is fit to produce code for a safety-critical system. These requirements are culled from our own work, previous work in provably-correct compilers, and from informal code-generation approaches. They are designed to satisfy ethical, legal, and regulatory concerns with using code generation. First, the source and target languages must be formal. Second, a formal meaning-preserving translation must be described between the two languages. Third, the implementation of the translator must be verified to confirm it implements the formal transformation. Fourth, the translator must be treated as high-assurance software and tested accordingly. Finally, the output from the compiler must be readable and easily traceable back to the original specification.

In order to validate these guidelines as sufficient, we are building a code generator suitable for safety-critical systems for the specification language RSML. The target language, SIMPL, was designed as a strict subset of many imperative languages used for development of safety-critical systems. By using SIMPL, we can simplify the equivalence proofs for the formal translation and also target multiple implementation languages in a separate, very small, and easy to implement translator.

References

[1] B. Auernheimer and R. A. Kemmerer. RT-ASLAN: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 12(9), September 1986.

[2] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[4] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[6] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[7] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[8] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[9] D.J. Keenan and M.P.E. Heimdahl. Code generation from hierarchicl state machines. In *Proceedings of the International Symposium on Requirements Engineering*, 1997.

[10] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[11] Program Validation Ltd. *Formal Semantics of SPARK*. Program Validation Ltd., 1998.

[12] British Ministry of Defence. *Requirements for Safety Related Software in Defence Equipment Part 1: Requirements*. British Ministry of Defence, 1991.

[13] RTCA. *Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[14] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.

[15] Susan Stepney. *High Integrity Compilation*. Prentice Hall, 1993.

[16] Susan Stepney. Incremental development of a high integrity compiler: Experience from an industrial development. In *Proceedings of the IEEE High Assurance Systems Engineering Workshop*, 1998.

[17] Steve Vestal. Assuring the correctness of automatically generated software. In *AIAA/IEEE Digital Avionics Systems Conference*, volume 13, pages 111–118, 1994.

[18] P. Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3), March 1991.