

CAREER: Runtime Verification: Integrating Formal Methods Into Common Software Development Practice

David Coppit

PROJECT SUMMARY

Most software developed today exhibits a surprisingly low level of dependability. Formal, mathematically precise methods for specifying and implementing software are one approach to solving this problem. Despite much progress in formal methods, applying them to the development of provably correct software remains infeasibly costly for nontrivial systems. Instead, developers rely on testing as a much less rigorous, but also less costly, method of verifying the correctness of software.

The work described in this CAREER proposal seeks to integrate formal methods into software development practice through a combination of research and education. The goal of the research component is to develop *runtime verification* as a means of providing correctness guarantees similar to those of formal methods at a cost on par with testing. The goal of the educational component is to produce software engineers who have the training to effectively use formal methods in software development.

Intellectual Merit

Runtime verification involves the automatic generation of a monitor from a software specification. The monitor verifies the correct operation of the software by checking that the specification is not violated. This work will address three key research problems which must be overcome before this approach can be practical.

The first problem is that we lack an understanding of the structural and semantic gaps which exist between specification and programming languages. A precise characterization of these gaps is crucial for the automatic generation of software monitors. Without this understanding, the necessary correspondence between elements in the specification and implementation cannot be established. We propose to analyze and characterize this mismatch in the context of several different specification and implementation languages.

The second problem is that we do not fully understand the design space for languages to support runtime verification. We expect that many unnecessary structural and semantic gaps can be attributed to the independent design of specification and implementation languages. We propose to investigate co-design of the languages as a means of bridging these gaps. As part of this investigation, we hope to be able to precisely characterize the nature of the correctness guarantees provided by runtime verification.

Finally, we do not yet know how to integrate runtime verification into software development practice. We lack an understanding of the impact of this approach on today's software development methodologies. We lack comprehensive, easy-to-use tools for writing specifications, generating software monitors, and verifying implementations using those monitors. We also lack data on the cost-effectiveness of this approach as applied to real software development efforts. We propose to develop tools to support runtime verification, to disseminate these tools to practitioners, and collaborate with practitioners to evaluate this approach.

Broader Impact

This proposal seeks to address the still unsolved problem of developing correct software at a reasonable cost. The research seeks to make the benefits of formal methods widely available to software practitioners by developing and disseminating cost-effective tools and techniques for runtime verification. At the same time, the educational component of this proposal complements the research by seeking to provide software engineers with the knowledge necessary to apply such mathematically rigorous methods to the development of software. In addition, runtime verification can have a significant impact on software development activities such as implementation, testing, and reliability assessment.

CAREER: Runtime Verification: Integrating Formal Methods Into Common Software Development Practice

David Coppit

PROJECT DESCRIPTION

Formal methods are mathematically precise notations, tools, and techniques used for the development of software. The use of formal methods is widely considered to be expensive, despite evidence that their use can result in software systems with fewer faults [42]. The centerpiece of formal methods is the use of a specification of the software expressed in a mathematically precise notation. Such a specification can be validated informally by expert inspection, or formally using tool-assisted theorem proving techniques. The resulting specification is usually more precise, unambiguous, and complete than an informal natural-language specification.

The next step, creating a software implementation that correctly implements the specification remains a difficult task. Historically, formal methods were developed to help prove that particular implementations are correct [19, 37]. Another more recent approach is to refine the specification into an implementation using a series of semantics-preserving transformations [43]. Both of these approaches can result in software that is correct for all possible inputs. Unfortunately, these techniques are too expensive to apply for all but the most critical of systems. In fact, formal methods are widely considered to be prohibitively expensive to apply in practice.

Instead, *testing* continues to be the most widely used method of ensuring the dependability of software. Each test case subjects the software to a set of inputs, and verifies that the output is correct. Unfortunately, testing is only a partial proof of correctness in that it does not guarantee that the system will operate as expected under untested inputs [18]. This problem is exacerbated by the discontinuous nature of software—rare, unanticipated combinations of inputs often cause failures in deployed software [52].

In this research we will develop notations, tools, and techniques for *runtime verification* of software. Runtime verification has many of the benefits of formal methods, but at a cost that is on par with testing. The basic idea is to develop a formal specification of the software, then use the specification to automatically generate a monitor which can verify the correct operation of the implementation at runtime. Runtime verification is cheaper than traditional formal verification because *checking* software correctness for particular inputs is usually much easier than *proving* software correctness for all possible inputs. Employing runtime verification in practice is also cheaper because the checks implemented within a monitor can be derived from the specification.

In terms of its ability to guarantee software correctness, runtime verification is weaker than formal methods but stronger than testing. Testing can only guarantee the correctness of a limited set of inputs at implementation time. As a result, undiscovered faults may result in failures at runtime, sometimes allowing the system to continue to propagate corrupted output because the failure was not detected. By always monitoring the software for correctness, such failures can be caught when they happen, for any input which causes them to occur. However, runtime verification is weaker than formal method because such guarantees can not be made *a priori*.

1 Scientific Objectives

Some of the necessary elements for this approach have already been developed: formal languages exist for the specification of software, and assertions are used during development to check the correct operation of code. However, we are a long way from being able to automatically generate comprehensive runtime monitors with well-understood guarantees of correctness.

The following simple example will help to make the concept of runtime verification more concrete, and will be used to illustrate the scientific objectives of this proposal. Consider an algorithm to sort a list of integers:

```
void Sort(int numbers[], int array_size) {
  for (int i = 0; i < array_size; i++) {
    int save = numbers[i];
    for (int j = i ; j > 0 && (numbers[j-1] > save) ; j--)
      numbers[j] = numbers[j-1];
    numbers[j] = save;
  }
}
```

Formally proving the correctness of this algorithm is beyond the capabilities of most programmers. Furthermore, proofs of algorithms such as this are not reusable in the sense that a different implementation of the function would require a new proof. And for large, complex software systems it would be infeasibly costly to develop such correctness proofs.

However, the specification for the algorithm is fairly succinct and straightforward. The following specification is written in Z [48], a popular software specification language which is based on typed set theory and first-order predicate logic.

$$\left| \begin{array}{l} \textit{Sort} : \textit{seq} \mathbb{Z} \rightarrow \textit{seq} \mathbb{Z} \\ \hline \forall \textit{in}, \textit{out} : \textit{seq} \mathbb{Z} \bullet \textit{Sort}(\textit{in}) = \textit{out} \Leftrightarrow \\ \textit{items}(\textit{in}) = \textit{items}(\textit{out}) \wedge (\forall i, j : 1 \dots \#\textit{out} \mid i < j \bullet \textit{out}(i) \leq \textit{out}(j)) \end{array} \right.$$

The portion of the specification above the horizontal line is the type signature. It states that *Sort* is a total function which takes a finite sequence of numbers as its input and computes a finite sequence of numbers as its output. The portion below the horizontal line describes the behavior of *Sort*. For any given *in* and *out* sequences, *out* will be the result of applying *Sort* to *in* if and only if two conditions hold. The first is that *out* is a permutation of *in*. The second is that for every two positions *i* and *j* in the *out* sequence where *i* is less than *j*, the integer at position *i* in the *out* sequence must be less than or equal to the integer at position *j*.

Once such a specification has been developed, it must be validated to ensure that it is correct and complete. There are a number of approaches to this task. Type checking provides a simple check that is similar to syntax checking a program. Model checking allows the user to simulate the specification in order to ensure that certain theorems hold, while theorem proving proves theorems via symbolic manipulation of the specification.

Once the specification has been validated as correct, it is possible to translate it into embedded assertions in the implementation. The following implementation illustrates the idea. (Some functions are not shown for brevity.)

```
void Sort(int numbers[], int array_size) {
  int *old_numbers = copy_array(numbers, size);

  for (int i = 0 ; i < array_size ; i++) {
    int save = numbers[i];
    for (int j = i ; j > 0 && (numbers[j-1] > save) ; j--)
      numbers[j] = numbers[j-1];
    numbers[j] = save;
  }

  assert(Is_Permutation(old_numbers, numbers, size);
}
```

```

for (int i = 0 ; i < size - 1 ; i++)
    assert( numbers[i] <= numbers[i+1] );
}

```

If an execution of this function does not fail an assertion, then the result is *guaranteed to be correct* with respect to the specification. Unlike formal proofs of correctness, the monitor is easy to write, and is not dependent on the particular implementation of the function.

This approach is more powerful than the simple `assert` mechanism sometimes employed by programmers, in that the embedded monitor provides more comprehensive validation of the result rather than simple implementation-dependent “spot checks”. It is also more powerful than testing, which can only address particular inputs. This approach can trap failures at runtime and reveal faults for rare and unexpected states which were not addressed during testing.

While it is possible to manually translate specifications into runtime monitors, this is a tedious and error-prone task. Ideally, a developer would use a tool to automatically derive monitoring code from a specification and embed it into the implementation. There are a number of research challenges which must be overcome before this is feasible.

1. We lack an understanding of the structural and semantic gaps which exist between specification and programming languages.

Before specifications can be automatically transformed into program monitors, there must exist a mapping from structural elements in the specification to those in the code. This mapping must take into account differences in the levels of abstraction in the two languages, as specifications do not address many of the details addressed by the implementation. In the *Sort* example, a structural correspondence exists between the specification function and the implementation function. However, the developer could have easily replaced the inner loop with a call to a separate function. Such structural gaps between corresponding elements at the implementation and specification levels complicate automatic generation of monitoring code.

Even when the correspondence between elements can be identified, semantic gaps may exist. In the *Sort* example, the input is defined as a sequence of integers ($\text{seq } \mathbb{Z}$), but implemented as an array and size parameter. In this case, the unprotected bounds of the array may compromise the trustworthiness of the checker, which assumes that the code can not modify data outside of the sequence. Similarly, the monitor implementation requires a separate `Is_Permutation` function because the language does not support the “bag” data type and *items* operation used in the specification.

2. We lack an understanding of the design space of languages for runtime verification.

One reason for the gaps between specification and programming languages is that the languages, to date, have been designed independently. It should be possible to help bridge this gap through co-design of both languages with runtime verification in mind. For example, the *Sort* implementation could be based on a data structure library which would replace the array with a polymorphic `seq` data structure that would correspond to the `seq` data structure of the specification language. Conversely, the specification language could be modified to remove infinite data structures which can not exist in an implementation.

3. We lack a full understanding of the nature of the guarantees provided by runtime verification.

It is already known that certain types of specifications can not be verified at runtime. For example, liveness properties (that “something will eventually happen”) can not be verified because the monitor can only check the current and past state of the software. There are also limitations resulting from the manner in which specifications are expressed. State-based specification languages such as Z do not

allow the developer to easily express temporal properties, and temporal languages such as LTL [23] are generally poor at expressing the structure of state. Specification languages also do not directly address memory limitations and other implementation details which can cause software to fail.

These limitations beg the question of what “correct” really means. Understanding the limitations of runtime verification will allow researchers to better characterize software that has been verified as correct by a monitor. In the *Sort* example, it would be desirable to add a constraint to the specification that states that the algorithm is $O(n \log n)$. However, Z does not support such statements, and it is not clear how the software would be monitored to check such a constraint even if it did.

4. **We do not yet understand the role and impact of runtime verification on broader software engineering activities.**

We do not yet know the relationship between runtime verification and the software development life-cycle. For example, we do not know how this approach integrates with current specification or verification techniques. Most importantly, we have no data on the benefits relative to the costs, without which the approach is unlikely to be adopted into practice. And the current dearth of easy-to-use, high-quality tools makes such an evaluation by real users impossible.

2 Impact

Runtime verification promises to be a practical, low-cost approach to improving the quality of software. It attacks the most important and still elusive problem in software engineering—that of building correct software at a reasonable cost. Runtime verification is a specification-first approach that forces the developer to think precisely about *what* the software should do, instead of *how* it should be done.

Because the software is always verified by the runtime monitor, the approach has the potential to change the way users think about software—they can have much higher confidence in the software than traditional testing can provide. By augmenting runtime verification with automatic error reporting, end-users can also participate in the development process by identifying subtle faults which can only be found in a complex “real world” environment. The benefit of this approach is similar to the “given enough eyeballs, all bugs are shallow” phenomenon observed in the open-source community [44], except that the user need not be a software developer, and the source need not be available.

In the development of highly reliable software, developers augment testing with statistical analysis in an effort to compute a quantifiable estimate of the reliability of the software. In this field, imperfect error detection is a significant risk [1]—errors which are not detected can result in artificially high reliability estimates. Runtime verification can provide a means of largely eliminating undetected errors, resulting in reliability estimates which have much better confidence intervals.

There are also intriguing consequences for traditional testing. Testing is hampered by the need to independently compute or verify the expected outputs for each test case. By testing with runtime verification, testers do not need to derive expected outputs because the monitor will certify the output as correct. This can greatly reduce the cost of testing, and enable a “continuous testing” approach in which the software is tested against automatically generated inputs until the monitor fails. This approach has the potential of significantly increasing the effectiveness of testing because testers are no longer limited to small sets of test cases with validated outputs.

Finally, the pragmatic, cost-effective nature of this research will help to ensure that the resulting techniques, languages, and tools can be more easily adopted into engineering practice. In addition to the publication of scholarly papers, tools that are developed will be released to the public in an effort to foster their use and build a community of users. This community will help to evaluate the approach in a non-academic context, and will likely reveal new research opportunities. The educational component of this proposal seeks

to provide students with the intellectual tools necessary to take advantage of formal methods in software development. It will also introduce students to this and other innovative research in formal methods.

3 Prior Research

My research to date has been on cost-effective techniques for the development of high-quality software tools. The approach I have explored involves advances in two areas: applied formal methods and component-based software development techniques. In this section I describe my previous research and relate it to this proposed research.

3.1 Development Methods for Software Tools

Software tools are widely used in many disciplines to develop and analyze models of real-world phenomena. The analysis results are then used to infer the behavior of the phenomena. For example, aeronautical engineers use computer simulations to model the behavior of aircraft under conditions which can not be reproduced with a wind tunnel. Software engineers develop UML diagrams [6] to model data relationships and module interactions in software.

In order for tools to be used safely and effectively, several necessary conditions must hold:

- **The modeling language must be well-defined.**

Without a precise definition of the modeling language, users of the tool may misunderstand the meaning of models they develop, and there is no documented basis for evaluating the correctness of the implementation.

- **The tool must be of high quality.**

In order for a tool to be used effectively, it must support a high degree of usability and a wide range of functionality which users have come to expect. Implementing such usability and functionality is usually costly. For example, implementing an “undo” feature requires that every operation have some way to return to the previous state. Low quality tools expose the user to additional risk because there is an increase chance that the user will unknowingly misuse the tool.

- **The implementation must be trustworthy.**

For a tool to be dependable, its implementation must be trustworthy. This is especially true for the implementation of the modeling language and analysis algorithms which are central to the tool. It would be irresponsible and perhaps even life-threatening to for a user to trust the results of poorly implemented tools without independent validation.

- **Development costs must be kept low.**

Building dependable, high-quality tools for well-defined modeling languages is very costly using traditional approaches. As a result, many tools are expensive and compromise in one or more key dimensions.

My research has resulted in a tool development approach which is based on the combination of two techniques: formal methods to aid in the design and validation of the modeling language, and an innovative component-based development method to provide the features and usability of tools at low cost [11, 14]. In this approach, formal methods are applied in a judicious way to the definition of the critical modeling language. The less-critical but nevertheless costly supporting functionality is provided by specializing and

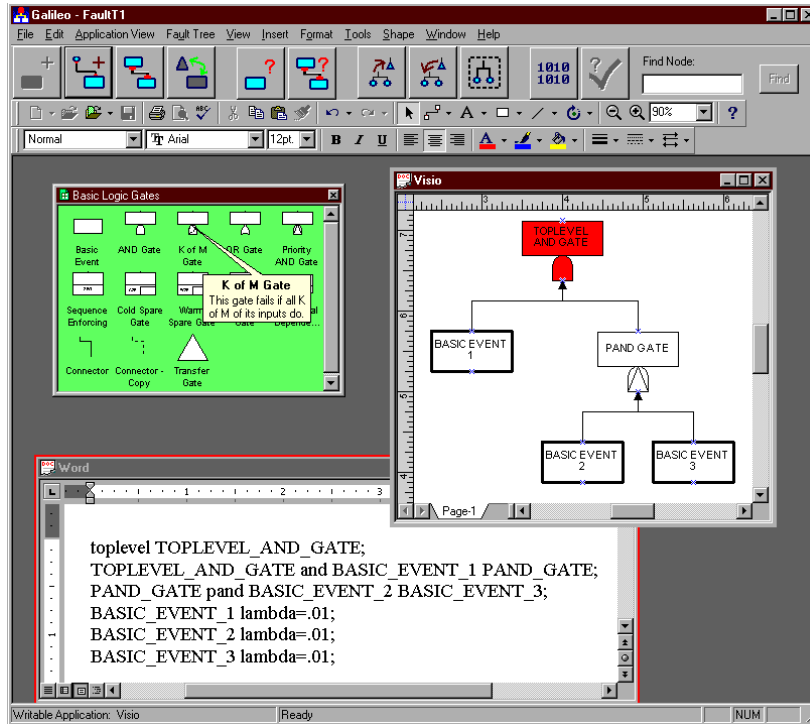


Figure 1: A screenshot of Galileo

integrating mass-market applications as components in a style called *package-oriented programming* [13, 50].

This approach has been evaluated in an end-to-end experiment in which it was used to build a tool called *Galileo* [12, 21, 51]. Galileo is a reliability modeling and analysis tool which supports the dynamic fault tree language [7, 20]. A screenshot of the tool is shown in Figure 1. The bulk of the functionality of the tool is provided by highly customized and integrated mass-market applications—Microsoft Word and Visio. These applications allow the user to edit and analyze dynamic fault tree models in an easy-to-use and richly functional user interface.

As part of the evaluation of the approach, engineers from across NASA were invited to use Galileo in three workshops. Surveys taken at the workshops indicate that Galileo’s usability and features meet or exceed that of commercial tools. The enthusiastic feedback from participants at the workshop has led to the adoption of Galileo for use by a NASA group working on the International Space Station. Overall, this research has resulted in an approach to developing tools which can help produce feature-rich, highly-usable tools supporting formally-defined languages at a greatly reduced cost.

3.2 Toward Runtime Verification

The tool development approach described above is only a partial solution because it does not address the the dependability of the implementation. In fact, it was this issue which was the impetus for investigating runtime verification as a cost-effective means of addressing this problem. In this sense, the proposed work will help complete a key missing component of my previous research in tool development methods.

As part of my previous research, I evaluated the cost-effectiveness of several aspects of formal methods. I found that the initial development and informal validation of the specification were very useful, revealing significant errors at a modest cost. This result is encouraging because it suggests that the high cost traditionally associated with formal methods is largely due to formal proofs of code and/or formal refinement of

specifications to code. Runtime verification requires neither of these steps, which indicates that the approach can work.

Part of this research will involve the development of tools to support runtime verification. In this regard, I believe that my prior experience in tool development methods will be of some use. Providing easy-to-use tools is essential for properly evaluating the approach from the perspective of real users, and will eventually help to disseminate the results of this research into practice. My prior collaborations with reliability engineers are also likely to provide interesting case studies for evaluating this approach.

4 Research Plan

This section describes the plan for executing this proposed research. It is organized into four phases. This research will focus on state-based specification languages, as opposed to temporal specification languages. The latter will be addressed in subsequent work.

4.1 Assessment of the State of the Art

The first phase of this research is devoted to assessing the current state of the art and planning the details of the remaining work. There are two purposes for this endeavor: to become intimately acquainted with the strengths and weaknesses of approaches that are currently in use, and to identify candidate languages and tools to serve as the starting point for work in later phases. Several languages and tools have already been identified, and some preliminary work has already begun.

In order to evaluate existing languages, tools, and techniques, three case studies will be conducted in which several different formal specification languages are used to specify software systems. Specifications will be manually translated into runtime monitors using a number of enabling implementation technologies. Then the monitored implementations will be evaluated using existing test suites, exhaustive testing, and fault injection.

4.1.1 The Case Studies

The first case study will involve the Z [48] specification language and the C++ programming language. This part of the evaluation will leverage my previous work in reliability engineering, as part of which I developed, validated, and implemented a formal specification of the dynamic fault tree modeling and analysis method. Validation was based on simple type checking using the fuzz [49] and ztc [33] tools, as well as the proof of theorems about the specification using Z/Eves [46]. The C++ implementation was structured in a manner similar to the specification, anticipating the eventual inclusion of manually-implemented runtime monitoring code.

This work has already revealed some insight into the deficiencies of the Z language and its supporting tools. For example, the lack of object-oriented support in Z caused the specification and implementation to be structured in an awkward manner. As expected, type checking was confirmed to be easy to apply but provide little validation of the specification. Theorem proving was found to require a high degree of expertise and computational power, and to require a large up-front investment before interesting theorems could be proven.

What still remains to be done is to manually translate the specification into a monitor, and then to evaluate the effectiveness of the monitor in identifying faults. There are two possible avenues for implementing the monitor. The first is to simply implement the monitor using the C `assert` mechanism. The second is to use the somewhat richer assertion capabilities of a compiler such as the Digital Mars compiler [39]. As part of the fault identification process, I plan to take advantage of ongoing collaborative work with the University of Virginia. As part of this work, we have developed a method of exhaustively enumerating possible test inputs

from an input specification. This approach will be used to generate inputs which will drive the verification effort.

The second case study will involve the Object-Z [47] specification language and the Java programming language. This work is already in progress, and involves an existing component of the Test Driver Generator software developed by Cigital, Inc. [10] A student is currently developing a formal specification of the component using Object-Z. This specification will be validated informally with developers of the component and more formally via type checking and specification animation. (There is currently no usable theorem prover for Object-Z.)

Because the implementation of the software already exists, this case study will provide the opportunity to evaluate this approach for software which was not designed with runtime verification in mind. Our preliminary work indicates that the object-oriented nature of Object-Z is an improvement over Z, but there are still significant language limitations. For example, it is difficult to express the semantics of operations which are defined as compositions of other operations.

The Jass [3] tool will be used to aid the manual implementation of the runtime monitor. Jass extends Java to provide a richer assertion capability than the default assertion in Java, including quantification, preconditions and postconditions, and inheritance of class invariants. Our strategy for evaluating the effectiveness of the monitoring is to use fault injection, comparing the successful identification of the fault using runtime monitoring versus a traditional test suite.

The third case study has yet to be determined. One possibility is the Parallel Graphics Library [16], a graphical rendering system whose distributed nature would add a new dimension to the evaluation. The primary author of that system works here at the College of William and Mary. A second possibility is a job scheduling and monitoring system used by physicists at the nearby Jefferson labs. This system is interesting in that it contains a number of programmer-created assertions which can be used to help evaluate specification-derived monitoring code. A third possibility which I am pursuing is to build a collaboration with the nearby NASA Langley Research Center in order to acquire access to a suitable software system in the aerospace field.

4.1.2 Evaluation Criteria

The purpose of performing the case studies is twofold. The first goal is to understand the capabilities and limitations of today's languages and tools with respect to their appropriateness for runtime verification. The second is to identify and characterize key obstacles to the overall approach, and to use this knowledge to help plan the remaining details of the research effort.

In terms of specification languages, the case studies will help us to gain an understanding of the expressibility of the languages and the impact of this expressibility on the feasibility of efficient runtime verification. A key question is whether a given specification language allows the developer to easily express the semantics of the software system. On the other hand, it is possible that a language is *too* expressive, allowing specifications to be written which can not be checked by a software monitor. For example, a proposition which involves a universal quantification over the set of integers ($\forall x \mid x \in \mathbb{Z} \bullet P(x)$) can not be expressed using a computer's finite arithmetic. Another key goal is to identify language features which adversely affect monitoring performance. Examples include universal quantification statements which have been constrained to a finite but exceedingly large set of potential values.

For the implementation languages, the primary goal is to identify semantic and structural gaps with respect to the specification. The case studies should also reveal opportunities for raising the level of abstraction in implementation languages to better support automatic generation of runtime monitors. For example, data structures in the C++ Standard Template Library (STL) support a standard iterator mechanism which can greatly ease the translation of quantification in the specification into iterator-based loops in the implementation. Similarly, while the STL supports the `map` data structure, it does not have a native `function` class

with associated operators for performing domain restriction, extracting the range, etc. Libraries can also be developed to bridge the gap by providing such specification language abstractions.

The case studies will use existing tools such as Jass in order to determine the extent to which the more advanced assertion capabilities can help bridge the gap between specification and implementation. The aim is avoid “reinventing the wheel”—to build upon work that has already been done by learning the design decisions of existing tools and by seeking out collaborations with the researchers responsible for their development.

In terms of the overall approach, case studies will also help identify technical difficulties resulting from the manual implementation of monitors. These insights will help guide our subsequent efforts in the development of an automated approach. The case studies will allow us to compare the implementation and integration of monitors derived from specifications prior to and after implementation. This work will also provide additional data on the effort required to develop and validate a formal specification. A careful study of the performance of instrumented code will help reveal bottlenecks, potential tradeoffs, and opportunities for optimization.

4.2 Co-Design of Specification and Programming Languages

The insights and experience gained from the first phase of the research will aid the co-design of specification and implementation languages in the second phase. The goal of this phase is to enable automatic translation of specifications into runtime monitors through the careful design of the languages.

Because specification languages are still a topic of research, there is some freedom to create new languages or modify existing ones. Of the available state-based languages, Object-Z seems to be the most suitable basis for the development of a specification language with good support for runtime verification. It is similar to the more widely used Z, but adds support for writing specifications in an object-oriented manner.

It is likely that changes to the specification language will limit its full mathematical generality in order to ensure that specifications written in the language can be translated to efficient software monitors. Restricting the language in this manner may not overly constrain its expressiveness. For example, Jackson [30] has developed Alloy, a specification language whose expressiveness has been limited to enable efficient model checking. Case studies have shown that Alloy, while limited, still allows useful specifications to be developed and validated.

From a practical standpoint, it is unlikely that much progress can be made by developing new programming languages to support runtime verification. Many organizations have made large investments in training, compilers, and tools for standard programming languages such as C++ and Java. A more promising approach that has been taken by other researchers is to adapt an existing language. AspectJ [35], for example, was initially implemented as a pre-processor which generated standard Java code, and now has full compiler support. AspectJ now has a large user base, due at least in part to the decision to implement aspect-oriented programming as an addition to standard Java.

Another approach is to use the programming language’s built-in customization support to raise the level of abstraction. In this regard, C++ may be a better language than Java, with its support for features such as templates and overloaded operators. (Note, though, that some version of these features is planned for future versions of Java.) The language can be specialized to disallow pointer manipulation, support specification-oriented data structures and operations, and to even support automatic memory management [5].

Once the language design is complete the original case studies will be reworked (again, manually) to use the new languages. This will help to validate our designs and will reveal any previously undiscovered obstacles to automatic generation of program monitors.

4.3 Development of Supporting Tools

Once the initial language design is complete, the third phase will involve the development of tools to support the overall method. In keeping with the goal of fostering practical adoption, this research will focus on extending popular integrated development environments (IDEs) to support the approach.

There are several components of the planned tool support:

1. Specification Editor

The IDE must provide a means of editing specifications. There are two possible approaches. The first is to require that the user write the specification separately from the source code. This approach allows the developer to use non-ASCII characters such as “ \forall ” which are common in formal specifications. It also encourages the developer to develop and reason about the specification independently of the implementation. However, it does complicate the correlation of specification elements and implementation elements, perhaps requiring the developer to specify a mapping between the two.

Another approach is to allow the user to embed formal specifications in the source code, as done in languages which support design-by-contract [41]. As described later in Section 6.3, this approach can tempt the developer to write only partial, implementation-oriented specifications. It also requires that the specification be extracted from the implementation for independent validation. However, because the specification is embedded in the implementation, the correlation between the two may not need to be explicitly defined by the developer.

2. Validation Support

Tools must also provide assistance validating the specification. The goal is to provide validation tools which are as simple to use as a compiler, and which require little if no assistance from the developer. This strategy is essential for maintaining the cost-effectiveness of this approach.

There is some evidence that lightweight validation approaches can provide significant benefit over more costly approaches such as theorem proving. Type checking is a minimal but easy-to-apply validation method. Model checking of a limited subset of the overall state space has also been shown to be surprisingly useful [32]. Finally, recent work [25] has shown that specification-based static analysis of source code can be both useful and automated if one is willing to sacrifice some soundness in the analysis. In a similar way, it may be possible to validate specifications using automated analyses which compromise soundness, occasionally exhibiting fault positives or false negatives.

3. Automatic Generation of Monitor Code

A central component of the tool support is the generation and embedding of monitoring code into the software. This process should be fully automated, and an invisible component of compilation process. There should be no change in the semantics of the program. Indeed, the only observable change in the (correct) execution of the software is a performance penalty. Runtime monitoring should interact cleanly with debugging of the program, and generated object code containing monitoring code should be link-compatible with object code without the monitoring code.

Probably the most straightforward manner of providing this capability is to implement it as a pre-processor which generates modified source code which can be compiled by a standard compiler. Compiler front-ends such as those developed by the Edison Design Group [22] have the capability to do source-to-source transformation of code. Another approach is to modify an existing compiler to support “compilation” of specifications into monitoring code. The GNU compiler suite [53] is a likely candidate should this avenue be taken.

4. Automated Testing Support

The final component in the IDE will be automated testing support. This will allow the developer to write a specification which describes the form of the input. This specification will be used by a testing tool to automatically generate inputs to drive the verification of the software during testing.

This is the approach briefly described in Section 4.1.1 and implemented by the Alloy analyzer [31]. The input specification is translated into a Boolean expression which is submitted to a standard SAT solver. The SAT solver returns solutions in the form of assignments of Boolean values for the variables in the expression. These solutions can then be “un-translated” to provide instances of valid inputs. In this way, it is possible to exhaustively enumerate the inputs in an increasingly large size.

The initial toolset and following versions will be released to the public. It is hoped that these tools, combined with scholarly papers, will encourage others to experiment with the approach. The feedback from users will help to identify potential improvements in the languages and tools for future versions, and will likely reveal new and interesting research directions.

4.4 Evaluation and Interaction with Other Activities

The final stage of this research is to perform an in-depth evaluation of the approach, and to assess its interaction with other software engineering activities. It is hoped that at this point the public release of the tools will have fostered collaborations which will provide a means of evaluating the approach in the context of real software development activities. This sort of evaluation is very useful in that it often reveals unanticipated insights often missed by researchers who perform evaluations using smaller, less realistic systems. It also provides the opportunity to study the approach in the hands of non-experts, and to study the impact of the approach on specification, implementation, testing, and other software development activities.

Even with external collaborations, we will also perform a more technical evaluation of the approach by using it in the development of several nontrivial software systems. For example, an empirical study of the effectiveness of runtime verification compared to traditional testing approaches would be useful. Here I plan to take advantage of my prior relationships with the reliability engineering community, where having the ability to detect faults with a high degree of accuracy is important in the testing of highly reliable systems.

This stage of the research will also involve a theoretical analysis of the ability of runtime verification to guarantee the correctness of software. The structural and semantic gaps between specification and implementation languages should be fairly well understood, if informally, at this point. The development of a theoretical framework for reasoning about the correctness of verified software seems well-suited for a PhD student, as it requires a substantive knowledge of formal methods.

5 Educational Plan

As an educator, my goal is to provide students with the intellectual tools necessary for reasoning about, designing, and developing complex software systems. There are two key obstacles to this goal. First, we are largely failing to following Fred Brooks’ advice to nurture great designers [8]. Second, many students lack the mathematical background necessary to approach software development from a mathematically rigorous point of view.

5.1 Nurturing Great Designers

The first problem is a result of curricula that focus on the mechanics of programming, and not in higher-level design. As a result, I have observed that many students “program by accident”—they repeatedly edit

poorly designed code until it finally compiles and passes one or two test cases. More often than not, the resulting software fails for other inputs. This type of education fosters a reliance on testing as a partial proof of correctness, when a more abstract approach can help produce code which is correct for all inputs.

At the College of William and Mary, we are already taking steps to help solve this problem. We are restructuring the curriculum around Java instead of C++. It is believed that Java is a simpler language, the mechanics of which can be taught more quickly. This will provide some freedom to focus on higher-level design. To this end, we are introducing a new sophomore-level software design and implementation course. I will be one of two faculty members who will be developing this course. I hope to teach students how to decompose a small software system into modules, how to design module interfaces, and how to implement software in a modular “always a working system” manner.

In my senior-level software engineering course, I will reinforce their design education with architectural-level design. The students will learn the major architectural styles, and well as the characteristics of software developed for particular application domains. Examples of the latter include real-time software, event-driven software, distributed software, and safety-critical software.

5.2 Producing Software Engineers Skilled In Formal Methods

The second problem is partly the result of the United States educational system, where students are exposed to less mathematics than students in other countries. This has a direct effect on the success of formal methods in software engineering. For example, formal methods originated in Europe and continue to be widely used there. I believe this is due at least in part to the more substantial mathematics education foreign students receive. A second barrier to the use of formal methods is that they are widely believed to be too costly to apply. I believe that this is a result of teaching formal methods from a primarily theoretical rather than a practical point of view. As a result, students see little relevance or applicability of formal methods to their own software development efforts.

To address these issues, I plan to develop a new junior-level course in the application of cost-effective formal methods to software engineering. The course will help students become proficient in a formal specification language. It will demonstrate the benefits of using formal methods to specify and analyze the design of complex software systems. It will also introduce students to an assortment of tools which can be used to validate the specifications they develop.

The course will emphasize the cost-effective, practical application of formal methods. My own research in runtime verification will be an exemplar of such an approach. I plan to have students develop a formal specification of a nontrivial software system, then use my tools to translate this specification into a monitor. The students will then execute the monitored software to identify faults. In addition, students will experiment with other approaches such as traditional testing and specification-based testing, comparing those approaches to runtime verification.

I expect that not only will my research benefit my teaching, but also vice-versa. I will be able to observe non-experts as they use the runtime verification approach, which will reveal difficulties which they encounter and opportunities for improvement. It may be possible to use this course to perform a more rigorous evaluation of the approach. Feedback from the students will help to determine the extent to which this approach is truly cost-effective.

6 Background

6.1 Program Checking and Monitoring

The basic concept of checking program results at runtime is not new. As early as 1967 assertions have been used in a systematic manner to help verify software [26]. Several authors have advocated developing software

with checkers to help detect invalid program executions, and perhaps even take corrective action [4, 28, 55].

There are three key distinctions in the work we propose. The first is that software is verified against a documented and validated specification, without which developers are likely to reproduce conceptual errors in both the code and the checker. (Typically, assertions are used to verify *implementation* issues and not *specification* issues.) The second is that runtime verification encourages the development of an overall specification prior to implementation, which helps to ensure that subtle interactions between components can be identified. Lastly, this research seeks to be able to automatically generate monitors from specifications, instead of assuming that the software developer will shoulder that burden.

In more recent work [9, 27, 29, 36] programs are monitored to ensure the correctness of temporal logic properties. In this approach, the code is instrumented to send events to the external monitor. The performance issues with this approach limit the monitoring capability to partial specifications expressed in the form of constraints.

One goal this research is to help identify new techniques for runtime verification which are efficient and yet provide well-understood correctness guarantees. Initially, the research will focus on state-based specification languages, which have weaker support for expressing temporal properties. The scope may be extended later to support temporal logic specifications, depending on the results of our characterization of the effectiveness of the correctness guarantees provided by the state-based approach.

6.2 Static Analysis

Static analysis seeks to locate dynamic memory faults at compile time. Typically static analysis approaches augment the code with additional information about the behavior of data at interface points. The Splint tool [25, 24], for example, allows the user to embed partial specifications of the software into the code, and then uses this information to help identify bugs such as buffer overflows and invalid pointer usage.

ESC Java [17] uses a similar approach that allows the user to write specifications of functions, then uses an automatic theorem prover to reason about the correctness of the implementation. Microsoft's Slam [2] extracts a model from the source code, and uses a model checker to ensure that the source code interacts correctly with the operating systems, whose behavior is formally specified.

Static analysis approaches can be viewed as complementary to runtime verification. The goal of these approaches is to attempt to identify faults before deployment of the software. These methods are necessarily limited in that they must infer or simulate abstractly the runtime behavior of the system. Runtime verification can check the actual runtime behavior of the system, but at a cost—performance is degraded, and faults are detected after deployment.

6.3 State-Based Assertion Approaches

There have been a number of research projects which seek to increase the power of simple assertions. Perhaps the most widely known is Meyer's design-by-contract [41]. Design-by-contract is a lightweight approach to embedding specifications in the form of assertions in object-oriented code. Originally a part of the Eiffel programming language, design-by-contract has been extended to Java [3, 34], Ada [38], and many other languages.

Contracts are a structured, object-oriented approach to traditional assertions. Contracts specify preconditions and postconditions on methods as well as class invariants. Contracts can be inherited. For example, refinements to the contract of an overloaded method can only weaken the precondition and/or strengthen the postcondition. Contracts are used primarily during development to help ensure the correct behavior of interacting modules, and are often disabled at runtime for efficiency.

This proposed research differs in two key ways. First, this research advocates the independent development of a specification. A key problem with design-by-contract is that the contracts are not separately

validated from the implementation, which forces developers to conduct “mini proofs” by hand to ensure the correct composition of specifications. Independent specifications ease informal validation and enable tool-assisted formal validation via model-checking, theorem proving, and other well-known techniques. Second, this research seeks to make runtime verification more efficient, thereby removing the need to disable checks at runtime.

6.4 Testing

Some work has been done in the integration of runtime verification and testing. Voas and Miller [54] observe that assertions can help testing by localizing faults within the code, and detecting when internal state is incorrect even though the output is correct. Their work seeks to use sensitivity analysis to identify locations where assertions are most likely to identify faults.

There are two tacit and unproven assumptions in their work. First, they assume that comprehensive verification of the implementation requires assertions at every program location. In runtime verification, monitoring code need only be placed to check the results of implementation components which correspond to specification-level abstractions. Second, their work assumes that verification is very costly. However, Rosenblum reports [45] no discernible speed difference between code containing many assertions and code containing none. In this research, We hope to demonstrate that comprehensive verification is possible with little performance overhead.

Crane and Dingel [15] have developed an approach to checking the conformance of an implementation which uses a model checker to verify that the state of the running system is valid. The program is periodically stopped, and its state is converted via an abstraction mapping into an instance of the specification. The model checker then analyzes the instance to determine whether it conforms to the specification.

The approach taken in this research seeks to automatically embed the verification into the implementation. The result will be a more efficient, portable program, the verification of which does not require communication with external applications. Their approach is also limited by the specification language of the model checker, and does not address the structural gap which arises as a result of different levels of abstraction between the specification and implementation. This research seeks to address both of these issues.

Finally, some work has already been done on the translation of specifications into test oracles. McDonald and Strooper [40] recognize the difficulty of developing oracles which can provide correct outputs for the development of test cases. Their approach is similar to the one proposed here—Object-Z specifications are converted into C++ test oracles which are used to verify the output of the program at runtime.

The work to date, however, has several limitations which this proposed research seeks to address. First, their mapping between the specification and the implementation is a simple name-based scheme which does not address varying levels of abstraction. Second, the approach has not yet been demonstrated to be efficient enough to run on nontrivial software systems. Third, the authors have begun to evolve the the specification language by adding support for exceptions, but more work remains in this regard. Perhaps most importantly, the translation of the specification is manual—obviously a serious limitation of the work.

7 Summary

Runtime verification is a promising approach to significantly increasing the quality of the software we produce, at a modest cost. While some work has been performed in this area, we are a long way from being able to automatically generate monitors for software. I propose to develop new languages, tools, and techniques to help reach that goal, and to understand the theoretical limitations of the correctness guarantees provided by the approach. At the same time, my teaching plans seek to produce software developers who are equipped with the necessary skills to take advantage of this and other formal approaches to developing software.

CAREER: Runtime Verification:

Integrating Formal Methods Into Common Software Development Practice

David Coppit

REFERENCES CITED

- [1] Paul E. Ammann, Susan S. Brilliant, and John C. Knight. The effect of imperfect error detection on reliability assessment via life testing. *IEEE Transactions on Software Engineering*, 20(2):142–8, February 1994.
- [2] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103–22, 2001.
- [3] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – Java with assertions. In Klaus Havelund and Grigore Roşu, editors, *Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 23July 2001.
- [4] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–91, January 1995.
- [5] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–20, 2000.
- [6] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusettes, 1999.
- [7] Mark A. Boyd. *Dynamic Fault Tree Models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems*. PhD thesis, Duke University, Department of Computer Science, April 1991.
- [8] Fred Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.
- [9] Mark Brörkens and Michael Möller. Dynamic event generation for runtime checking using the JDI. In Klaus Havelund and Grigore Roşu, editors, *Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 26July 2002.
- [10] Cigital, Inc. TDoG internal documentation, 1998.
- [11] David Coppit. *Engineering Modeling and Analysis: Sound Methods and Effective Tools*. PhD thesis, The University of Virginia, Charlottesville, Virginia, January 2003. URL: <http://www.cs.wm.edu/~coppit/papers/dissertation.pdf>.
- [12] David Coppit and Kevin J. Sullivan. Galileo: A tool built from mass-market applications. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 750–3, Limerick, Ireland, 4–11 June 2000. IEEE.
- [13] David Coppit and Kevin J. Sullivan. Multiple mass-market applications as components. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 273–82, Limerick, Ireland, 4–11 June 2000. IEEE.
- [14] David Coppit and Kevin J. Sullivan. Sound methods and effective tools for engineering modeling and analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 198–207, Portland, Oregon, 3–10 May 2003. IEEE.

- [15] Michelle L. Crane and Juergen Dingel. Runtime conformance checking of objects using Alloy. In *Runtime Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, Boulder, Colorado, 13July 2003. Elsevier. URL: <http://www.cs.queensu.ca/home/stl/papers/CraneDingel-RV03.ps>.
- [16] Thomas W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23(7):819–43, July 1997.
- [17] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report #159, Compaq Systems Research Center, Palo Alto, CA, December 1998. URL: <ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-159.ps.gz>.
- [18] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.
- [19] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [20] Joanne Bechta Dugan, Salvatore Bavuso, and Mark Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–77, September 1992.
- [21] Joanne Bechta Dugan, Kevin J. Sullivan, and David Coppit. Developing a high-quality software tool for fault tree analysis. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 222–31, Boca Raton, Florida, 1–4 November 1999. IEEE.
- [22] Edison Design Group. Compiler front ends for the OEM market, 2003. URL: <http://www.edg.com/>.
- [23] E. A. Emerson. *Temporal and Modal Logics*, chapter 16. Elsevier, 1990.
- [24] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53, San Diego, CA, 21–24 May 1996.
- [25] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96, New Orleans, USA, 6–9 December 1994.
- [26] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, New York, New York, 5–7April 1967. American Mathematical Society.
- [27] A. Q. Gates, S. Roach, O. Mondragon, and N. Delgado. DynaMICs: Comprehensive support for runtime monitoring. In Klaus Havelund and Grigore Roşu, editors, *Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 23July 2001.
- [28] Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. *Tool Support for System Specification and Verification*, chapter Programs without Proof? On Checker Based Program Verification, pages 108–23. Springer Series Advances in Computing Science. Springer Verlag, London, UK, 1999.
- [29] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. In Klaus Havelund and Grigore Roşu, editors, *Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 23July 2001.
- [30] Daniel Jackson. Alloy: A lightweight object modelling notation. URL: <http://sdg.lcs.mit.edu/~dnj/publications.html>.

- [31] Daniel Jackson. Automating first-order relational logic. In David S. Rosenblum, editor, *Proceedings of the Eighth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 130–9, San Diego, CA, 6–10 November 2000.
- [32] Daniel Jackson and Kevin Sullivan. COM revisited: Tool assisted modelling and analysis of software structures. In *Proceedings of the Eighth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 149–58, San Diego, CA, 6–10 November 2000.
- [33] Xiaoping Jia. ZTC: A type checker for Z. notation user’s guide. URL: <http://se.cs.depaul.edu/fm/ztc.html>.
- [34] Murat Karaorman and Parker Abercrombie. jContractor: Introducing design-by-contract to Java using reflective bytecode instrumentation. In Klaus Havelund and Grigore Roşu, editors, *Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, pages 56–80. Elsevier, 26 July 2002.
- [35] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.
- [36] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-MaC: a run-time assurance tool for Java programs. In Klaus Havelund and Grigore Roşu, editors, *Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 23 July 2001.
- [37] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley, 1979.
- [38] David C. Luckham and Friedrich W. von Henke. Overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–224, March 1985.
- [39] Digital Mars. Digital Mars C/C++ compiler for Win32. URL: <http://www.digitalmars.com/>.
- [40] Jason McDonald and Paul Strooper. Translating Object-Z specifications to passive test oracles. In *2nd International Conference on Formal Engineering Methods*, pages 165–74, Brisbane, Queensland, Australia, 9–11 December 1998. IEEE. URL: <http://www.computer.org/proceedings/icfem/9198/91980165abs.htm>.
- [41] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [42] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–80, May 1979.
- [43] Carroll Morgan. *Programming from Specifications*. Prentice Hall, Hempstead, UK, 2nd edition, 1994.
- [44] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O’Reilly & Associates, Inc., 1999.
- [45] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
- [46] Mark Saaltink. The Z/EVES system. In *ZUM ’97: Z Formal Specification Notation. 11th International Conference of Z Users. Proceedings*, pages 72–85, Berlin, Germany, 3–4 April 1997. Springer-Verlag.

- [47] Graeme Smith. *The Object Z Specification Language*. Kluwer Academic Publishers, 1999.
- [48] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
- [49] Mike Spivey. The fuzz manual. URL: <http://spivey.oriel.ox.ac.uk/~mike/fuzz/>.
- [50] K. J. Sullivan and J. C. Knight. Experience assessing an architectural approach to large-scale systematic reuse. In *Proceedings of the 18th International Conference on Software Engineering*, pages 220–229, Berlin, Germany, 25–30 March 1996. IEEE.
- [51] Kevin J. Sullivan, Joanne Bechta Dugan, and David Coppit. The Galileo fault tree analysis tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 232–5, Madison, Wisconsin, 15–18 June 1999. IEEE.
- [52] Dong Tang and Herbert Hecht. An approach to measuring and assessing dependability for critical software systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 192–202, Albuquerque, New Mexico, 2–5 November 1997. IEEE.
- [53] The Free Software Foundation. The GCC home page. URL: <http://gcc.gnu.org/>.
- [54] Jeffrey M. Voas and Keith W. Miller. Putting assertions in their place. In *Proceedings of the International Symposium on Software Reliability Engineering*, Monterey, CA, 6–9 November 1994. IEEE.
- [55] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–49, November 1997.