High-Level Reliability Languages Using A General Intermediate Domain

Robert R. Painter, M.S., Dept. of Computer Science, The College of William and Mary
David Coppit, Ph.D., Dept. of Computer Science, The College of William and Mary

Key Words: intermediate modeling languages, formal specification, reliability engineering, Markov models

SUMMARY & CONCLUSIONS

The increasing complexity of today's systems has led reliability engineering researchers to develop high-level modeling languages with sophisticated modeling capabilities. Today, researchers develop high-level languages *independently* and in a *semi-formal* manner. As a result, researchers expend redundant effort in language design, and in the implementation and verification of the language in the form of analysis tools. The resulting languages have imprecise semantics, and are difficult to compare because they have no common semantic basis.

In previous work [1] we argued that an intermediate language can significantly ease the difficulty of formalizing and implementing reliability languages. We showed that a formally-defined intermediate language can provide a mathematically precise abstraction upon which one can define new high-level modeling languages. In this paper, we present an improved and generalized version of our intermediate language which we call the failure automaton (FA). By utilizing a common formal semantic domain such as the FA, languages have a precise meaning, the cost of developing and implementing languages is reduced, advances can be more easily shared among languages, and understanding the differences between languages is easier.

To test the generality of the FA, we used it to define and implement the semantics of three dependability modeling languages. Our new approach to the definition and implementation of reliability modeling languages helps find ambiguity, reduces implementation effort, and allows sharing of language features.

1. INTRODUCTION

The increasing complexity of today's systems has led reliability engineering researchers to develop high-level modeling languages with complex semantics. High-level languages are intuitive because they have been designed using domain-specific syntax and semantics. That is, engineers can express models using notations and meanings that are familiar to them. For example, the spare gate in the dynamic fault tree language is used to model the use and failure of spare components in systems. The spare gate idea can be well-understood by practitioners.

Unfortunately, the complex interactions between seemingly simple high-level language constructs such as the spare gate are difficult to understand. Traditional approaches to defining reliability modeling notations have been *semi-formal*. A semi-formal definition has a well-defined syntax but an imprecise or incomplete semantic definition. For example, the syntax may be specified formally in a mathematical notation such as Backus-Naur form (BNF) grammars, but the semantics may be only illustrated by showing the precise meaning for a set of examples. Semi-formal definitions can lead to ambiguous semantics and untrustworthy tools, because such limited examples can not capture the subtle interactions between modeling constructs which occur in general.

Secondly, today's reliability modeling languages are developed *independently*. Consider the following three reliability languages: reliability block diagrams (RBDs), dynamic fault trees (DFTs), and Boolean-Driven Markov Processes (BDMPs). Researchers are actively developing these languages, which are higher-level and have semantics can be expressed in terms of Markov models. The mapping of DFTs and BDMPs to low-level Markov models is difficult. This difficulty arises both in the *design* of the language and its *implementation*. As new reliability languages are developed, researchers must complete the difficult semantic mapping from the high-level language to the low-level language. Many times, researchers must waste time and effort re-solving problems that were previously overcome in the semantic definition of other languages.

Overall, the independent, semi-formal development of reliability languages can lead to high cost, semantically ambiguous, and poorly understood modeling languages. *We propose the use of a formally-defined common intermediate reliability modeling language to address this problem.*

Figure 1 illustrates our approach. High-level languages are mapped to a common intermediate language. The high-level language mappings allow reuse of the intermediate language's mappings to low-level languages. An intermediate language also helps reduce the overall design and implementation cost

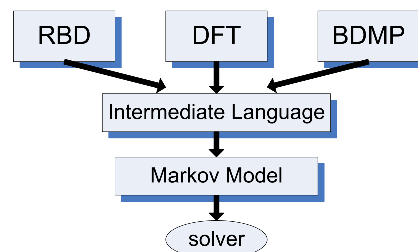


Figure 1: High-level language mappings to an intermediate language.

by reducing the "distance" from the high-level language to the low-level models.

A formal intermediate language benefits many parties. Because the intermediate language is formal, designers can have increased confidence in it, and can use it as a target for formal definition of the high-level language. Engineers can also have higher confidence in the consistency and meaning of the language. Furthermore, because a formal specification of the language exists, developers can use it to increase the dependability of the implementation. Researchers and users can also begin to analyze differences and similarities between languages.

While initial design, development, and verification of formal specifications is costly, this cost is amortized as the intermediate language is reused.

To evaluate this approach, we generalized an intermediate language which we had previously developed [2] called the Failure Automaton (FA). In order to assess the generality of the language, we have used it to formally specify three reliability modeling languages: Reliability Block Diagrams (RBDs), Dynamic Fault Trees (DFTs), and Boolean Driven Markov Processes (BDMPs). Finally, we investigated the benefits and costs associated with the use of an intermediate language for implementing the high-level languages.

Our experiments revealed that an intermediate language can be used to separate concerns in the definition of high-level languages. We found that the use of our intermediate language has eased the initial implementation costs. We implemented solvers for each of the three high-level languages using an FA solver as a library. We reduced the amount of code required by reusing the FA solver library. We also found that the specification is a good reference for understanding what we are programming.

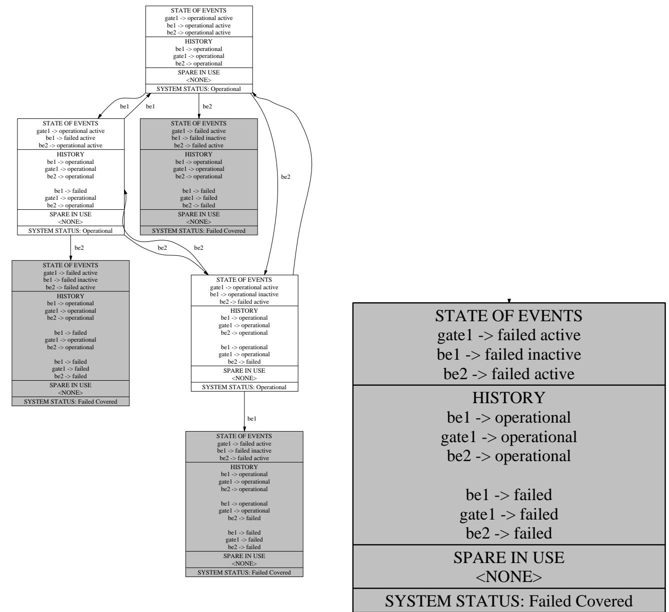
The primary contribution of this paper is a demonstration that it is possible to develop a general intermediate language that supports sophisticated, high-level reliability modeling concepts such as repair. As secondary contributions, we present our generalized failure automaton as an instance of such an intermediate language, and show how it can be used to formally define and implement three high-level reliability modeling languages.

The rest of this paper is organized as follows. Section 2 describes our intermediate language approach in more detail and describes the origins of the FA in the specification and implementation of DFTs. An overview of the formal specification of BDMPs is provided in Section 3. Our implementation and project evaluation are described in Section 4 and Section 5. Section 6 contains related and previous work. Section 7 concludes.

2. THE FAILURE AUTOMATON

The Failure Automaton (FA) was originally developed as an intermediate step in the formal specification of the DFT language [2]. The FA was formally defined, making it a sound basis for expressing the semantics of DFTs. However, it was a fault tree specific intermediate domain unsuitable for specifying the semantics of other high-level languages. For example, the FA didn't support features that were foreign to DFTs such as failure-on-demand and repair.

Despite these shortcomings, we believed that the FA could be evolved to express many features common in high-level reliability languages. Therefore, we modified the FA to create a general language capturing the features expressed in most high-level reliability languages. The version of the FA here introduces features such as repair and failure or repair on relief or demand, and refactors previous modeling capabilities such as event replication and basic event models.



(a) Overview (b) Detail

Figure 2: An example failure automaton

2.1 FA Overview

The FA is a state-based language that is structured like Markov models (MM) except reliability model information is contained in the states and numerical aspects such as transition rates are abstracted. An example FA is shown in Figure 2. In the overview (Figure 2(a)), the transitions between states are labeled with the events that caused the change in state, which we call *causal basic events*. A transition is associated with either a failure or a repair. In the former case, the destination state will show that the causal basic event has failed. In the latter case, the destination state will have an operational causal basic event.

In the figure we have shaded the system states which correspond to overall system failures. In the original model (not shown), changes in the system state occur as the result of events $be1$ and $be2$. In the associated FA, these events are the causal basic events for the transitions out of the *Operational* states shown in the overview figure. For example, from the start state at the top of Figure 2(a), there are three outgoing failure arcs.

Figure 2(b) shows the state on the second row, right side in detail. The information inside each state represents the current state of events, history, spares in use, and system status. The state of each event consists of a *failure status* and an *activity status*. Failure status indicates whether the event has occurred, and activity status indicates whether components are idle.

The history records past failures. This is necessary for order-dependent high-level language constructs like sequence enforcers or Priority AND (PAND) gates. The history is a record of failure order. The current failure status of events is appended to the history. The history always increases in size when a failure occurs. A repair occurs when a basic event goes from a failed to operational state. When a repair occurs the component is considered to be repaired *as if never failed* [1], so that all past failures of that event are erased from the history. The history never increases in size as the result of a repair.

Spare-in-use is used to keep track of which of its inputs a spare gate is using. This is useful, for example, in determining spare allocation in DFTs. The system status indicates whether or not the system is failed and whether it failed as a result of normal *covered* failure or a catastrophic *uncovered* failure.

2.2 Basic Event Models

In the FA, events can either be basic events or derived events. Basic events are events that can occur on their own. The state of derived events depends on the state of a set of basic events and other derived events. In high-level models, derived events are often represented as gates. Ultimately, the state of the system depends on the behavior of the basic events and how they interact with each other as defined by the derived events. The FA models interaction of events but does not capture the individual basic event properties.

A Basic Event Model (BEM) describes the stochastic behavior of a basic event. Four distributions in the BEM describe the rate that an event transitions out of its current state. *activeDistribution* describes the behavior of an event that is not failed and is active. *inactiveDistribution* describes the behavior of an event that is not failed and is inactive. *repairActiveDistribution* describes the repair behavior of an event that is failed and is active. The *repairInactiveDistribution* describes the repair behavior of an event that is failed and inactive. BEM also contains a *coverage model* and *transfer function model*. A coverage model is a set of probabilities used to describe when an event occurrence has catastrophically caused a complete system failure. A transfer function model is a set of probabilities for FOD, FOR, ROD, and ROR.

The separation of FA and BEM has two key advantages. First, there is a conceptual separation of concerns between the derived event interactions described in FA and the basic event models described in BEM. As a result, the transitions of the FA do not have any rates associated with them. This conceptual simplification greatly reduces the distance between the high-level language and the FA because the FA is more abstract. Second, in practice, a large FA that is difficult to compute doesn't have to be recomputed when the basic event properties change.

2.3 FA Specification

The formal specifications are written in Z [3]. Z is a useful language for creating well-defined semantics because it is based on first-order predicate logic and set theory. The state of the system is described with schemas. Schemas allow the variables of the system to be declared and restrictions on the values of those variables to be placed.

In Z all variables must have a type. We defined the basic type that we use to represent an event as *Event*. We also define a *BasicEventModel* and map basic events to them using a *BasicEventModelFunction* function. We have omitted these definitions to save space.

FailureAutomaton

$$\begin{array}{l}
states : \mathbb{F}\, FailureAutomatonState \\
failureTransitions : \mathbb{F}\, FailureAutomatonTransition \\
repairTransitions : \mathbb{F}\, FailureAutomatonTransition \\
transitions : \mathbb{F}\, FailureAutomatonTransition
\end{array}$$

$$\begin{array}{l}
states = \bigcup \{ t : FailureAutomatonTransition \mid \\
\quad t \in transitions \bullet \{ t.from, t.to \} \} \\
\langle failureTransitions, repairTransitions \rangle \text{ partition } transitions \\
\\
\forall fat : FailureAutomatonTransition \mid fat \in transitions \bullet \\
\quad ((fat.from.failureStatusOfEvents(fat.causalBasicEvent)) = failed) \\
\qquad \Leftrightarrow fat \in repairTransitions \land \\
\quad \neg ((fat.from.failureStatusOfEvents(fat.causalBasicEvent)) = failed) \\
\qquad \Leftrightarrow fat \in failureTransitions \\
\\
\forall fat : FailureAutomatonTransition \mid fat \in transitions \bullet \\
\quad ((fat.from.failureStatusOfEvents(fat.causalBasicEvent)) = failed) \\
\quad\; \Rightarrow (\forall fsoe : FailureStatusOfEvents \mid fsoe \in \operatorname{ran} fat.to.history \bullet \\
\qquad \neg ((fsoe(fat.causalBasicEvent)) = failed)) \land \\
\quad \neg ((fat.from.failureStatusOfEvents(fat.causalBasicEvent)) = failed) \\
\quad\; \Rightarrow fat.to.history = fat.from.history \frown \langle fat.to.failureStatusOfEvents \rangle
\end{array}$$

The schema *FailureAutomaton* specifies the abstract syntax of the failure automaton. It includes the components described earlier and the invariants. A formal definition of *FailureAutomatonState* and *FailureAutomatonTransition* is not included to save space. Transitions are separated into two sets, failure and repair. The length of the history increases across failure transitions, recording the failure that occurred. An events record of failure is removed across repair transitions. Repair "as if it never occurred" semantics prevent an infinite number of states in the FA. (While this is not a problem in the mathematics, it does cause complications in the implementation.)

We define *FailureAutomatonSemantics* as a function that maps a *FailureAutomaton* and *BasicEventModelFunction* into a *MarkovModel*. We have omitted the definition for space reasons. The definition ensures that the states and transitions in the FA and MM correspond. It also states that the initial MM state probabilities are correct, and that the rates of the MM transitions have been calculated with the appropriate Hazard function and scaled by the appropriate factors and probabilities.

2.4 Differences with Previous Versions of the FA

As mentioned earlier, an early version of the FA was designed and formally specified by Coppit, Sullivan, and Dugan [2]. The original FA specification did not support features such as repair and FOD, FOR, ROD, and ROR. However it did support the notion of replication. In this section, we present the most substantial of many differences between the original FA specification and our new FA specification.

[1] The choice to repair *as if never failed* is somewhat arbitrary. The semantics of repair may be changed later to accommodate a more realistic interpretation.

Replication is the idea that multiple indistinguishable basic events can be represented with a single basic event and the number of events it represents. It became clear that replication is a high-level language feature, and not an inherent characteristic of the FA. Leaving replication in the FA would have seriously complicated our addition of FA features. For example, it was impossible to determine that a replicate of a replicated basic event had failed on demand in early drafts of our specification. We also determined that replication can be handled more easily in an early step before mapping to an FA. For these reasons, our version of the FA does not support replication.

Failure-on-demand occurs when an event fails as soon as it is activated. FOD is represented in the FA by changes in failure status and activity status across a single transition such that the event goes from operational/inactive to failed/active assuming it was not the causal basic event of the transition. It is assumed that there is no way for an event state to switch across a single transition like this unless it was the causal basic event or FOD.

Activity status of events is a significant feature of FA states. Activity status for an event indicates whether the event should occur at a reduced rate. The original FA had the notion of dormancy of spares, a scale factor used to reduce the failure rate of unused spares. Spare dormancy is a feature of DFTs. However, sparing and dormancy can be treated more generally as separate features. This is evident in BDMPs, which support different activity modes but does not feature sparing. For this reason every event has an activity status.

3. HIGH-LEVEL RELIABILITY LANGUAGES

In order to test the generality of the FA as an intermediate language, we used it to define and implement three reliability languages. Earlier versions of the DFT and RBD definitions are published elsewhere [1, 2], and a more recent document contains updated versions [4]. In this paper, we will focus only on the formal specification of BDMPs.

3.1 Boolean-Driven Markov Processes

The Boolean-Driven Markov Process (BDMP) language is a dependability modeling formalism that is almost as easy to use as fault trees and has the capability to model sparing, functional dependencies, and repair [5, 6]. A BDMP consists of four components. The first is a multi-top coherent fault tree. The second is a set of triggers which model dependence between basic events. Each trigger is represented graphically as a dashed arc from a source event to a target event. The third is a pair of triggered Markov processes associated with each basic event. The triggered Markov processes can model repair in the system. Fourth, a set of transfer functions which describe how the current state is transferred from one Markov process to the other when the basic event is triggered.

Figure 3 depicts an example of a BDMP. The trigger from A to C is triggered when event A occurs. The result is a change in the mode of C from passive to not-passive. The example is incomplete because we have not defined the Markov processes or transfer functions for each basic event.

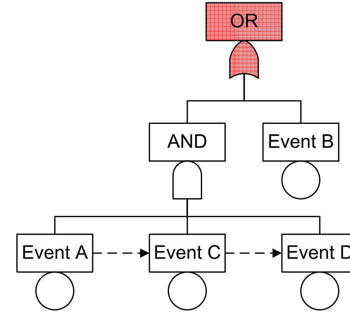


Figure 3: Example BDMP

3.2 BDMP Specification

In our specification we define the syntactic structure of BDMPs and the semantics in terms of the FA. We define a set of all *events* in the BDMP which is partitioned by basic events and gates. We also define an *inputs* mapping for the inputs of each gate in the BDMP. A set of constraints of BDMPs complete the structure by specifying only gates have inputs, no input loops exist, no two triggers can have the same target, etc.

The BDMP semantics are similar to that of fault trees. For example, AND gates are failed if all of their inputs are failed. The most interesting portion of BDMP semantics is the trigger semantics. The *TriggerSemantics* function specifies the semantics of triggering and implicitly FOD, FOR, ROD, ROR. Triggering occurs when a trigger origin's failure status is changed. The target of the trigger changes its activity status unless it is a gate. If the target of a trigger is a gate, the activity status switch cascades down through the inputs to the basic events.

4. IMPLEMENTATION

We implemented the FA language as a shared software library, and then used this library in the software implementation of RBD, DFT, and BDMP solvers. We created three solving tools for the three high-level languages: *Nova_solver* implements the DFT specification, *Firefly_solver* is an RBD implementation, and *BDMP_solver* is a tool for solving BDMPs. A solver library, called *fa_solver*, implements the FA. All of the solvers take a model written in a textual representation of the language, a language-specific BEM, and a mission time. The model is translated into an FA and the language-specific BEM is translated into a general BEM. These are passed to the *fa_solver* library which returns a Markov model.

In order to ease the verification of the code, we implemented it to follow the specification very closely. Therefore we did not optimize the code, or use object-oriented design because polymorphism and inheritance are not supported in Z.

Many functions and schemas translate directly into classes and methods. For example, the FA specification was easily written into code by closely following the document. We believe that this approach reduces the chance of implementation error—very few bugs have been found since its implementation. The ease in code writing was not seen in all implementation work. Because the specifications emphasize *what* and not *how*, some algorithm writing was difficult. For example, FOD transitions

were implicitly included in the BDMP specification. The specification considers all FA transitions and discounts those that did not meet the semantic restrictions of BDMPs. Implementing this would be infeasible. Therefore knowing how to construct only the valid transitions was difficult.

5. EVALUATION

Our approach had two key aspects to it. The first is a formal approach to the specification of high-level reliability languages. The second is the use of an intermediate language to reduce semantic gaps between language mappings. Evaluation of our approach is in terms of the effort involved in the specification and implementation of three high-level reliability languages.

5.1 Benefits of Intermediate Language

The use of an intermediate language reduced the level of understanding required when specifying the high-level languages. We were able to target an intermediate language that has a familiar domain-specific meaning. We learned that it is possible to use an intermediate language for a specific modeling domain to ease the specification of multiple high-level languages. This is because the semantic gap from high to low-level was reduced.

Our version of the FA captures many reliability model interpretations. We believe that the FA is an acceptable intermediate language for most high-level reliability model languages. However, one could argue that using the FA to define and implement three high-level languages is not enough proof of generality—other reliability languages may have unique features that the FA does not adequately support.

5.2 Insights into Intermediate Languages

After completing the specifications, we have a better understanding of what belongs in an intermediate language. For example, replication (Section 2.4) is more of a notation than a reliability model construct. We removed the replication from the FA and handled it in an early step of the DFT semantics. The specification of the FA and DFTs became better organized after removing replication. Even more important were the real limitations of the FA. For example, Failure-On-Demand couldn't be modeled in replicated basic events.

We learned that there are two concerns in reliability modeling: What are the behavior parameters of events occurring independently? How does the system depend on events? We handle these concerns separately until we have to join them: event behavior is modeled with the basic event model, and system dependence is modeled with the high-level language. The FA removed the burden of calculating and scaling hazard functions from the high-level language.

5.3 Coding Benefits

Use of a formal specification as a guide to writing code appears to be mixed in its benefits and costs. Some code was easy to write directly following a specification as in the implementation of *fa_solver* and *firefly*. For example, a key task of *fa_solver* is computing scaled hazard functions. The specification documented which scale factors to use and when. However, the specification did not always help. For example, it was difficult to ensure the correctness of code which implemented nondeterminism in the specification. This problem arose in the implementation of *Nova_solver* and *BDMP_solver*, both of which output FA models with nondeterministic transitions.

It was our experience that having a formal specification, while it did not provide guidance on *how* to implement the software, it did provide a clear standard for *what* was to be implemented. In particular, the definition of the FA as an intermediate representation made debugging easier because the FA is an easy-to-read notation that is full of rich semantic information. As a result, most debugging was done by reading the FA rather than obscure Markov models.

We believe that the *fa_solver* library saves the most implementation effort when used for the implementation of *multiple* high-level languages. Compared to another DFT solving tool [7, 8], the total lines of code related to the semantics of DFTs is approximately the same. However, the implementation costs for additional languages can be reduced because the *fa_solver* library can be reused. This belief has not been formally evaluated because we have not done an in-depth comparison of solver implementations performed both with and without the *fa_solver* library. There may be additional costs we have not considered, such as maintainability costs, which may be high if the FA and high-level languages are modified.

6. RELATED & PREVIOUS WORK

The work we propose is directly based on the work of Coppit, Sullivan, and Dugan [2] to develop a formal specification of dynamic fault trees. In their work, they cite many of the same problems with the design and implementation of reliability languages which we described previously. Our first efforts to use an intermediate language for multiple high-level languages utilized our earlier, DFT-specific version of the FA [1]. In that work we used the earlier FA to define and implement RBDs. This work extends our previous work, generalizing the FA as an intermediate language for more complicated reliability notations such as BDMPs.

Another language for developing reliability modeling languages is FIGARO [9]. FIGARO is a reliability modeling specification language used in the tool KB3 to develop high-level modeling languages. KB3 also allows the specification and solving of reliability models. While similar in an overall goal and approach, FIGARO differs from the FA. Figaro is a specification language like Z except it is reliability-modeling specific. FA is a modeling language intended to be the target of a semantic definition written in Z or FIGARO.

Malhotra and Trivedi [10] have developed a hierarchy of dependability models which includes fault trees, reliability block diagrams, and Petri Nets as well as others. To develop the power hierarchy, the authors developed algorithms to convert models expressed in one language into models expressed in another. Their work does not attempt to unite all the languages with a common intermediate language or develop the specification of

high-level languages.

Compilers such as GCC support a wide range of languages including C, C++, Fortran, Java, and Ada [11]. An intermediate language supports many languages easily and abstracts low-level hardware details. This work was inspired by the success of intermediate languages in programming language design and implementation.

7. CONCLUSION & FUTURE WORK

Languages differ in the trade-offs they make in terms of simplicity, high-level design and modeling power. Some are still developed in a semi-formal and independent manner. As a result, languages often have unclear semantics and are hard to compare, enhance, and implement.

We have tested the feasibility of using a common intermediate reliability language to address these problems. We developed an intermediate reliability modeling language and used it to formally defined the semantics of RBDs, DFTs, and BDMPs. We have also shown that the use of an intermediate language can help reduce the effort of initial high-level language implementation. Our approach also shows promise for easing language comparison and porting features between high-level languages.

In the future we hope to be able to show that our approach helps to answer the following questions. How does one language differ from another? What are the modeling limitations of a language? Can this language be extended to support more features? Does the implementation follow the specification?

ACKNOWLEDGMENTS

This work is based on previous work done in collaboration with Kevin Sullivan and Joanne Bechta Dugan at the University of Virginia. We would also like to thank Marc Bouissou for his help regarding the semantics of BDMPs and for commenting on an earlier draft of this paper.

REFERENCES

1. D. Coppit, R. R. Painter, and K. J. Sullivan, “Shared semantic domains for computational reliability engineering”, *Proceedings of the International Symposium on Software Reliability Engineering*, (Denver, Colorado), pp. 169–80, IEEE, 17–20 Nov. 2003.
2. D. Coppit, K. J. Sullivan, and J. B. Dugan, “Formal semantics of models for computational engineering: A case study on dynamic fault trees”, *Proceedings of the International Symposium on Software Reliability Engineering*, (San Jose, California), pp. 270–282, IEEE, 8–11 Oct. 2000.
3. J. M. Spivey, *The Z Notation: A Reference Manual*. Hertfordshire, UK: Prentice Hall International Series in Computer Science, 2nd ed., 1992.
4. R. Painter, “Toward a formal general intermediate dependability modeling language”, May 2004. unpublished.
5. M. Bouissou, “Boolean logic driven markov processes: A powerful new formalism for specifying and solving very large markov models”, *Proceedings of the 6th International Conference on Probabilistic Safety Assessment and Management*, (San Juan, Puerto Rico, USA), 23–28 June 2002.
6. M. Bouissou and J. L. Bon, “A new formalism that combines advantages of fault-trees and Markov models: Boolean logic Driven Markov Processes”, *Reliability Engineering and System Safety*, vol. 82, pp. 149–163, Nov. 2003.
7. D. Coppit and K. J. Sullivan, “Galileo: A tool built from mass-market applications”, *Proceedings of the 22nd International Conference on Software Engineering*, (Limerick, Ireland), pp. 750–3, IEEE, 4–11 June 2000.
8. K. J. Sullivan, J. B. Dugan, and D. Coppit, “The Galileo fault tree analysis tool”, *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, (Madison, Wisconsin), pp. 232–5, IEEE, 15–18 June 1999.
9. M. Bouissou, H. Bouhadana, M. Bannelier, and N. Villatte, “Knowledge modelling and reliability processing : presentation of the FIGARO language and associated tools”, *Proceedings of SafeComp’91*, (Trondheim, Norway), Nov. 1991.
10. M. Malhotra and K. S. Trivedi, “Power-hierarchy of dependability model types”, *IEEE Transactions on Reliability*, vol. 43, pp. 493–502, Sept. 1994.
11. The Free Software Foundation, “The GCC homepage.” URL: `http://gcc.gnu.org/`.

BIOGRAPHIES

Robert R. Painter, M.S.
Department of Computer Science
The College of William and Mary
P.O. Box 8795
Williamsburg, VA 23187-8795 USA

email: rrpain@cs.wm.edu

Robert Painter is a Ph.D. Candidate in the Department of Computer Science at the College of William and Mary. He completed his M.S. at William and Mary in 2003 and B.S. from Millersville University in 2001 with a Minor in Mathematics. His research interests include applied formal methods, algorithms, distributed systems, and anything with discrete math.

David Coppit, Ph.D.
Department of Computer Science
P.O. Box 8795
The College of William and Mary
Williamsburg, VA 23187-8795 USA

email: david@coppit.org

David Coppit is an assistant professor of Computer Science at the College of William and Mary. He received his PhD and MSCS degrees from the University of Virginia in 2003 and 1998. He received a BS in Computer Science and a BS in Physics from the University of Mississippi in 1995. His research interests include applied formal methods, software verification, and software representations.