# Automated Analysis of Software Frameworks (Position Paper)

Matthew B. Dwyer
Dept. of Computing and Information Science
Kansas State University
234 Nichols Hall
Manhattan, KS 66506 USA
Tel: (913) 532-6350
Fax: (913) 532-7353
Email: dwyer@cis.ksu.edu

**Abstract**

Domain-specific frameworks provide significant leverage for reusing design, implementation, validation and verification artifacts. The primary focus of most framework developers and consumers has been on leveraging implementations contained in a framework. Developers of high-assurance and safety-critical systems require more than just an implementation; they require very high degrees of confidence in the correctness of any software component they use. Traditional validation techniques, such as testing, cannot provide for such high-assurance.

In this paper, we discuss research issues in applying automated finite-state verification techniques to this problem. These techniques can be applied to verify properties of a framework without considering the details of how the framework is used in any particular application. The same techniques can, in turn, be used to verify that a framework is used correctly in an application. To do this requires the availability of precise partial specifications of properties of framework parameters. These parameter specifications allow verification of frameworks to be isolated from verification of applications, thereby enabling modularity in the verification process.

**Keywords:** Software frameworks, formal specification, static analysis, compositional and modular analysis, model checking

**Workshop Goals:** Develop a deeper understanding of the kinds of component-based reuse frameworks that are being developed, used, and researched. Open a dialog between the automated static analysis community and the frameworks community.

# 1 Background

We have conducted research on practical automated support for formal methods and on building domain-specific reuse frameworks for the past five years.

Our formal methods research focuses on developing and evaluating automated static analysis techniques, e.g., flow analyses and model checking, for verifying explicitly stated correctness properties of concurrent and distributed software systems. Our work is set in the context of event-based and propositional specification formalisms applied to safely approximating finite-state models of system behavior.

The primary barrier to practical application of these techniques is the well-known state-explosion problem, where the number of possible global system states grows exponentially in the number of system processes. Our work [DC94, DC97, DC96a, DCH97, DS97] uses abstraction techniques to enable reduction in the cost of analysis while preserving the ability to conclusively verify system properties. Of particular relevance to the analysis of component-based systems is our work on analyzing partial software systems [Dwy97]. In this work, we analyze portions of systems that are provided as designs or code along with partial specifications of the remaining system components. This offers a number advantages over traditional whole-system analysis techniques. It enables reduction in the time required to perform the analysis, by orders of magnitude in many cases. Perhaps more importantly, it allows developers to apply static analysis techniques earlier in the software development process. As soon as architectural descriptions, interface specifications or portions of implementations are available the verification and validation efforts can proceed. These interface specifications can be used both as the goals of unit-level formal verification and as components of sub-system and system level analyses.

Since a primary goal of our work is to evaluate the practicality of using formal specifications and verification techniques, we have spent significant effort in developing automated tools that support analysis approaches, for example the FLAVERS [Dwy95] and TPN [DC96a] toolsets. By applying theses tools to the analysis of formal specifications derived from requirements of realistic software systems we can begin to understand the pragmatic issues of scalability and performance that are crucial to wide-spread acceptance of automated formal methods.

An understanding of the potential for practicing engineers to adopt such methods cannot, however, be gained by measuring tool performance alone. We must also consider the non-trivial costs of learning specification formalisms and of gaining an understanding of the processes and strategies that allow one to read and write "realistic" specifications. During the course of our analysis work we have gathered experience with reading and writing specifications in both automata and logic-based notations. As a means of transferring the knowledge of experienced specifiers to potential users of formal methods, we have developed a pattern system for specifications of reactive systems [DAC97]. The hope is that just as design patterns have been effective in encoding and leveraging design knowledge, so too will specification patterns be effective for knowledge of formal specifications.

Our industrial experience in developing a family of optimizing compiler and debugger products established our appreciation for modular, component-based systems. We have continued to apply those principles in our research. We have analyzed and exploited information about three distinct software domains: flow analysis [DC96b], discrete-time grid simulations [DW97b, DW97a], and graphical user interfaces [DCH97]. In the first two of these we have defined and constructed parameterized frameworks that support construction of software systems in those domains. In our simulation framework, called PAGS, we have found the need for a broad collection of framework components that can be composed to fulfill commonly occurring application needs. Issues of performance as well as functionality are important in this domains. For these reasons our framework provides an open-implementation.

Our formal methods and software framework research are not completely independent. We have used our automated analysis techniques to verify properties of our frameworks and we are working on research that uses our simulation framework to implement parallel versions of analysis algorithms.

## 2 Position

### What is the problem?

Parameterizeable programming frameworks are becoming widely available to support developers in complex software domains. The strength of these frameworks is that they hide significant amounts of detail behind relatively simple interfaces. As with most software systems, validating a framework is typically done through a process of testing. Since a framework is not a complete application, it is necessary to construct a potentially elaborate collection of stubs and drivers to fill the roles of framework parameters and callers in order to test. Unlike for traditional applications, in testing frameworks the test designer may have little information about the potential behavior of framework parameters. Thus, one's ability to do a "good job" of testing may be reduced.

The problem for users of frameworks is determining whether the testing and validation efforts and results for a framework are representative of the context in which they wish to apply the framework. In other words, do traditional state-of-the-practice techniques for framework validation enable reuse and application of validation information in the context of systems built with the framework. Some researchers have addressed this problem for sequential reusable components by using formal specifications and developing proof methods that enable *local certifiability* of those components [WH94]. This approach takes the traditional formal methods approach of providing and reasoning about complete specifications of component behavior. In contrast, we believe that an approach based on partial specifications can yield a number of advantages. It offers the potential for practical fully-automated verification, via techniques such as model checking. It can address properties of both sequential and concurrent software frameworks; this is important because concurrent systems are very difficult to construct and to validate. It can be used to reason about a wide-range of correctness properties of systems (although there are properties that cannot be handled). We believe that an approach which adapts existing work on finite-state verification using the idea of local certifiability can bring the benefits of fully-automated verification without suffering from the well-known state explosion problem.

Such an approach decomposes framework validation into two steps:

1. Framework developers reason about correctness of frameworks in isolation

2. Framework users leverage those reasoning results for applications of framework

This decomposes the validation problem along boundaries that are similar to the design and implementation boundaries developers are familiar with. In this way it allows for separation of validation concerns. This approach raises a number of interesting research questions, which we present below.

### What is the significance for component-based software engineering?

Formal specification, even without the ability to analyze conformance of system behavior, offers a mechanism for succinct and precise descriptions of assumptions about the behavior of framework parameters as well as the capabilities provided by the framework. That those specifications might also be leveraged for testing, e.g., to derive test data or generate test drivers and stubs, or might be used as the goal of automated analyses is a bonus.

Traditional approaches to unit and sub-system level validation rely on testing. This is problematic for parameterized software frameworks since a framework intentionally makes minimal assumptions about parameter behavior and usage context. Thus, the problem of generating stubs and drivers that effectively excercise framework functionality can be a difficult one. Formal specification and automated analysis of frameworks provides an alternative validation technique that operates without difficulty in the presence of weak specifications and can, in fact, provide higher levels of confidence in framework correctness.

Component-based systems are defined in a modular fashion with explicit and well-defined boundaries. Scalable verification techniques must be designed to operate in two modes: they must allow verification within module boundaries and they must allow verification of module consumers without without *requiring* details hidden behind module boundaries.

Software frameworks are built to enable reuse. Typically, one thinks of the reuse of design or code information encoded in the framework. Another powerful type of reuse is possible, however, if formal specifications are bundled with framework components. We can leverage the efforts spent in validating framework specifications to applications built with those specifications. For example, we might verify that a framework is free of deadlocks if all of its parameters satisfy assumption $A$. We then need only analyze the parameters defined by an application to determine if they satisfy $A$, in which case, the instantiation of the framework is known to be deadlock free.

In this way, component-based systems can be validated early in the life-cycle, the specifications used in and results of those efforts can be bundled with components providing useful documentation and the opportunity to export knowledge about framework correctness to framework consumers.

## What insights might help solve the problem?

Success in addressing this problem will grow out of an understanding of the nature of realistic programming frameworks. This understanding can be used to drive the adaptation of existing specification and automated analysis techniques for the problem. In this position statement, rather than provide our own view of how the problem should be solved, we seek to stimulate discussion between framework researchers and researchers in automated analysis. Towards this end we lay out some research questions, with some thoughts on each question, in three areas: the nature of software frameworks, specifications of frameworks and analysis of frameworks.

## On The Nature of Software Frameworks

### What kinds of parameters are found in "real" frameworks and instantiations?

One can imagine that a very wide range of parameters are used to instantiate frameworks. Is this true? Are there common special cases, e.g., leaf routines, sequential code fragments?

### What are the space of parameter "usage semantics"?

There are lots of ways that components can be fed into a software framework, e.g., callbacks, value passing, procedure passing, object passing, inheritance, macro expansion.

How do the semantics of these mechanisms influence the incorporation of parameter descriptions into framework reasoning?

#### Do frameworks abstract concurrency?

If so, then we need to be prepared to handle a significantly more complex reasoning problem than in the purely sequential case. Furthermore, there may be a need for assessing *internal* correctness properties, e.g., deadlock freedom. The lightweight methods mentioned above all address concurrency to a certain extent. It may be the case that concurrency is present, but in a restricted form. Can the structure of this *internal* concurrency be exploited to reduce reasoning cost?

### Specification of Software Frameworks

#### What formalisms are appropriate/sufficient for describing framework behavior?

Frameworks can provide a wide variety of functionality. What are the tradeoffs in using different formal methods to describe those functionalities? Can lightweight methods, e.g., automata or propositional temporal logics, effectively describe critical components of behavior? Can hybrid approaches be used that separate light and heavy weight methods?

#### What formalisms are appropriate/sufficient for describing parameter behavior?

Just as frameworks provide functionality, they also require functionality from the components that parameterize it. Well designed frameworks make minimal assumptions about their parameters. In our experience these assumptions are much simpler to describe than the behavior of the framework itself. In general, are framework parameter assumptions amenable to encoding using lightweight methods?

### Analysis of Software Frameworks

#### Do stronger guarantees of parameter behavior enable stronger framework properties to be validated?

Frameworks make assumptions about their parameters. Practitioners of formal methods are free to model these assumptions in a variety of *safe* ways, i.e., by weakening the actual assumption. Usually we want to use the weakest possible assumptions because they provide the broadest degree of applicability for the reasoning result, i.e., those assumptions are satisfied in more contexts that use a framework. Given this range, can stronger parameter assumptions be put to good use?

**What analysis techniques can be effectively applied to frameworks and the components that parameterize and use them?**

Once we understand the range of formal specification methods that can be applied to frameworks and the components that interact with, then we want to understand what reasoning techniques are applicable. Can fully-automated techniques such as model-checking, flow analyses, and state-space search methods be applied to those portions of the problem for which lighter-weight methods are appropriate? Can these techniques be effectively integrated, where necessary, with reasoning techniques for heavier-weight methods?

**What is the cost vs. effectiveness tradeoff of framework analyses?**

This is really an empirical question. It should be clear that using an exact form of reasoning is infeasible. We are then left with performing an approximating analysis to reason about behavior; to the extent that such an analysis is *safe* or *conservative* we will be able to trust (some of) its results. Given this and considering the application of such methods to realistic software frameworks: Can limiting the scope of analysis to the framework effectively reduce validation cost? Do we actually reduce run time? Can limiting scope of analysis to the framework preserve sufficient levels of accuracy in their results? Do we get trustworthy results, i.e., that are as meaningful as a proof.

**How best can the results of framework validation and parameter assumptions be presented and reused by framework consumers?**

We could simply bundle specifications as text with the source code of a framework and satisfy the desire for precise documentation of frameworks. We could link parameter specifications with appropriate verification tools so that they could be invoked at framework instantiation time. What level of expertise will be required of consumers of this kind of capability? What about tools that require some kind of user interaction?

## 3 Comparison

The past decades of research on formal methods, modular systems and component reuse has set the stage for serious consideration of how these areas might best be integrated.

There is a wealth of research on program proving which served as the foundation for work on reasoning about data abstractions with respect to their specifications, for example the LARCH [GH93] and RESOLVE [SW94] efforts. These approaches have carried on in the tradition of using rich specification formalisms to give complete specifications of abstraction behavior. They, necessarily, use heavy-weight manual or automated approaches to reason about the conformance of implementations to those specifications. Both make the crucial observation that scalable formal reasoning is only feasible if it can be done in a local and modular fashion.

There has been a significant amount of research on techniques for reasoning about specifications of complete systems, e.g., [CGL94, McM93, YTL$^{+}$95, DBDS93]. There has been relatively little work on applying those analysis techniques to incompletely specified systems, e.g., parameterized frameworks and component-based systems, or in applying them in a way that achieves local certifiability [WH94].

The need for scalability of analyses and for analyses to integrate with practical development methods has shaped compiler analysis research to consider partial systems, for example as interprocedural flow analyses. Those same needs are beginning to shape the more general analyses targeted at software verification and validation.

Work on compositional analysis, e.g., [YY91, CA94, CK95, CK96], has, of necessity, dealt with the question of analyzing sub-systems. These techniques have the luxury of embedding such subsystem analyses in the context of a whole program analysis. In this setting specification of the behavior expected at a sub-systems interface can be automatically generated from the sub-system itself, e.g., by a process of hiding and partition-refinement. The work outlined in this paper, requires explicit specifications to be constructed. Consequently issues related to appropriateness, usability and exploitability of the specification formalisms come into play.

Some recent work has begun to look at issues relevant to the specification and analysis of component-based systems. Avrunin et. al. [ACD97] consider the analysis of systems that are defined in a mixture of implementation and specification languages. This approach allows the analysis of concurrent program components implemented in Ada along with missing system components specified with regular expressions or GIL [DKM+94] formulae. Two specification languages are used to address different issues in the specification of system and environment behavior; regular expressions capture ordering relationships and GIL formulae capture timing information. Mixed-formalism system descriptions are common in the early stages of all software development. They are particularly relevant to software frameworks, since we can never anticipate all of the potential parameters and usage contexts for a framework and must instead specify their requirements. While not a static analysis technique, recent work by Wang et. al. [WM97] explicitly considers the issues that arrise in validating parameterized software frameworks. The notion of symbolically executing a framework using abstract parameters is similar in spirit to our use of parameter specifications in exhaustive static analysis of framework behavior [Dwy97].

## 4 References

## References


[ACD97] G.S. Avrunin, J.C. Corbett, and L.K. Dillon. Analyzing partially-implemented real-time systems. In *Proceedings of the 19th International Conference on Software Engineering*, May 1997.

[CA94] J.C. Corbett and G.S. Avrunin. Towards scalable compositional analysis. *Software Engineering Notes*, 19(5):53–61, December 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CK95] S. C. Cheung and J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *ACM SIGSOFT '95 Symposium on Foundations of Software Engineering*, pages 140–150, Washington, DC, October 1995.

[CK96] S. C. Cheung and J. Kramer. Checking subsystem safety properties in compositional reachability analysis. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, March 1996.

[DAC97] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. Technical Report 97-10, Kansas State University, Department of Computing and Information Sciences, 1997.

[DBDS93] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz. Using state space methods for deadlock analysis in Ada tasking. *Software Engineering Notes*, 18(3):51–60, July 1993. Proceedings of the International Symposium on Software Testing and Analysis.

[DC94] M.B. Dwyer and L.A. Clarke. Data flow analysis for verifying properties of concurrent programs. *Software Engineering Notes*, 19(5):62–75, December 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[DC96a] M.B. Dwyer and L.A. Clarke. A compact petri net representation and its implications for analysis. *IEEE Transactions of Software Engineering*, 22(11), November 1996.

[DC96b] M.B. Dwyer and L.A. Clarke. A flexible architecture for building data flow analyzers. In *Proceedings of the 18th International Conference on Software Engineering*, March 1996.

[DC97] M.B. Dwyer and L.A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report 97-7, Kansas State University, Department of Computing and Information Sciences, 1997.

[DCH97] M.B. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In *Proceedings of the Sixth Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1997. to appear.

[DKM+94] Laura K. Dillon, G. Kutty, Louise E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[DS97] M.B. Dwyer and D.A. Schmidt. Limiting state explosion with filter-based refinement. In *Proceedings of the ILPS workshop on Verification, Abstract Interpretation and Model Checking*, October 1997. to appear.

[DW97a] M.B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency: Practice and Experience*, November 1997. to appear.

[DW97b] M.B. Dwyer and V. Wallentine. Object-oriented coordination abstractions for parallel software. In *Proceedings the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, June 1997.

[Dwy95] M.B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs.* PhD thesis, University of Massachusetts at Amherst, September 1995.

[Dwy97] M.B. Dwyer. Modular flow analysis for concurrent software. In *Proceeding of the 12th IEEE Conference on Automated Software Engineering*, November 1997. to appear.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[SW94] M. Sitariman and B.W. Weide. Component-based software using resolve. *Software Engineering Notes*, 19(4):21–22, October 1994.

[WH94] B.W. Weide and J.E. Hollingsworth. On local certifiability of software components. Technical Report OSU-CSIRC-1/94-TR-4, Department of Computing and Information Science, The Ohio State University, 1994.

[WM97] C. Wang and D.R. Musser. Dynamic verification of c++ generic algorithms. *IEEE Transactions of Software Engineering*, 23(5):314–323, May 1997.

[YTL+95] M. Young, R.N. Taylor, D.L. Levine, K.A. Nies, and D. Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):64–106, January 1995.

[YY91] W.J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification*, pages 49–59, Victoria, Canada, October 1991.

## 5 Biography

**Matthew B. Dwyer** is an Assistant Professor in the Department of Computing and Information Sciences at Kansas State University. He conducts research on specification and automated validation of correctness properties of concurrent and distributed software systems. He also researches frameworks to enable rapid development of reliable, high-performance software for scientific and computational engineering applications. In both of these areas, a key component of his work is the application and evaluation of research results in realistic settings. He received the Ph.D. in Computer Science from the University of Massachusetts in 1995. Prior to that he was a Senior Engineer at Intermetrics, Inc. for 5 years working on compiler and debugger product development. His work is funded by NSF under CAREER award CCR-9703094 and jointly with DARPA under EDCS award CCR-9633388.