Property Specification Patterns for Finite-State Verification[*]

Matthew B. Dwyer
Kansas State University[†]

George S. Avrunin
University of Massachusetts[‡]

James C. Corbett
University of Hawaii[§]

Abstract

Finite-state verification (e.g., model checking) provides a powerful means to detect errors that are often subtle and difficult to reproduce. Nevertheless, the transition of this technology from research to practice has been slow. While there are a number of potential causes for reluctance in adopting such formal methods in practice, we believe that a primary cause rests with the fact that practitioners are unfamiliar with specification processes, notations, and strategies. Recent years have seen growing success in leveraging experience with design and coding patterns. We propose a pattern-based approach to the presentation, codification and reuse of property specifications for finite-state verification.

1 Introduction

Formal specification and verification have been active areas of research for over two decades. While formal approaches offer practitioners some significant advantages over the current state-of-the-practice, they have not been widely adopted. In addition to a lack of definitive evidence in support of the cost-saving benefits of formal methods, a number of more pragmatic barriers to adoption of formal methods have been identified [26] including the lack of good tool support, expertise in organizations, good training materials, and process support for formal methods.

We believe that the recent availability of tool support for finite-state verification provides an opportunity to overcome some of these barriers. Finite-state

[*]This work was partially supported by NSF grants CCR-9308067, CCR-9407182, and CCR-9703094

[†]Department of Computing & Information Sciences, 234 Nichols Hall, Manhattan, KS 66506-2302. dwyer@cis.ksu.edu

[‡]Department of Mathematics & Statistics, Box 34515, Lederle Graduate Research Tower, Amherst, MA 01003-4515. avrunin@math.umass.edu

[§]Department of Information & Computer Science, Honolulu, HI 96822. corbett@hawaii.edu verification refers to a set of techniques for proving properties of finite-state models of computer systems. Properties are typically specified with temporal logics or regular expressions, while systems are specified as finite-state transition systems of some kind. Tool support is available for a variety of verification techniques including, for example, techniques based on model checking [23], bisimulation [8], language containment [18], flow analysis [15], and inequality necessary conditions [3]. In contrast to mechanical theorem proving, which often requires guidance by an expert, most finite-state verification techniques can be fully automated, thus relieving the user of the need to understand the inner workings of the verification process.

Despite the automation, users of finite-state verification tools still must be able to specify the system requirements in the specification language of the tool. For example, a user who wants to verify that "Between process A updating a value and process B reading the value, the value must be flushed from process A's cache" using a linear temporal logic (LTL) [21] model checker would have to translate this prose into the formula:

$$\Box((\mathit{UpdateA} \wedge \Diamond \mathit{ReadB}) \rightarrow \neg \mathit{ReadB}\ \mathcal{U}\ \mathit{FlushA})$$

Although the property and the formula are relatively simple, writing the LTL formula requires knowledge of several standard LTL idioms. For example, the property is (implicitly) a safety property, thus the formula begins with the $\Box$ operator. To say event Q always follows event P, we would usually write a specification of the form $P \rightarrow \Diamond Q$. In our case, however, we want to constrain the intermediate events, so we use the until operator $\mathcal{U}$ instead of $\Diamond$. Since the *FlushA* need not occur unless the *ReadB* does, we prohibit *ReadB* until *FlushA* only if a *ReadB* does eventually occur.

Even if they do not make use of all of the features and expressive power of the specification formalisms associated with formal verification techniques, users of those techniques do need to be expert enough to accurately express the requirements they wish to verify in the appropriate specification formalisms. We contend that acquiring this level of expertise represents a substantial obstacle to the adoption of automated finite-state verification techniques and that providing an effective way for practitioners to draw on a large experience base can greatly reduce this obstacle.

We propose to capture this experience base and enable the transfer of that experience between practitioners by way of a *specification pattern system*. Patterns were originally developed to capture recurring solutions to design and coding problems [17]. Design and coding languages are rich expressive formalisms that provide for a wide-variety of solutions to a given problem, but the full range of possible solutions is is usually much wider than is necessary or useful. Patterns are successful because practitioners want to solve naturally occurring domain problems. They don't need the full expressiveness of the languages they use and would often prefer guidance on how best to use language features to solve a specific problem. The same appears true in formal specification languages for concurrent and reactive systems. While there are a number of very expressive formalisms, such as CTL* and the modal mu-calculus, the specifications that are documented in the literature, for example in [1,6,12,14,24,27], appear relatively simple. They can be expressed fairly simply in existing specification formalisms and don't require advanced, complex features. Thus, we believe a collection of simple patterns can be defined to assist practitioners in mapping descriptions of system behavior into their formalism of choice, and that this may improve the transition of these formal methods to practice.

In the following section we describe the idea of design patterns and how that idea has been extended to software development domains other than design. Section 3 discusses the application of patterns to the description of specifications and lays out our terminology and format for describing specification patterns. Section 4 describes an initial specification pattern system for finite-state verification. We then evaluate and discuss the usefulness of the pattern system and mechanisms through which this system can mature and grow. Section 6 summarizes and concludes.

2 Design and Other Patterns

Design patterns were introduced [17] as a means of leveraging the experience of expert system designers. Patterns are intended to capture not only a description of recurring solutions to software design problems, but also the requirements addressed by the solution, the means by which the requirements are satisfied, and examples of the solution. All of this information should be described in a form that can be understood by practitioners so that they can identify similar requirements in their systems, select patterns that address those requirements, and instantiate solutions that embody those patterns. It is important to stress that not all descriptions of artifacts are patterns. Most design specifications and documents do capture a solution to a domain problem, describe requirements and provide an example solution, yet, they are not patterns. Patterns seek to generalize experience across multiple specific problems. Care must be taken, however, to keep patterns from being too abstract or removed from practice. A well-defined design pattern has the following characteristics [2]:

It Solves a Specific Problem, or class of problems, rather than being an abstract principle or strategy.

It is a Proven Concept that has been demonstrated to be effective in practice.

The Solution isn't Obvious and is not a direct application of basic principles.

It Describes Relationships between solution components rather than isolated components of a solution.

It is Generative in that it demonstrates how to construct a solution.

An active community has grown up around the idea of design patterns, as evidenced by the formation of numerous workshops (e.g., [10]) and recently the notion of patterns has been spreading to other software related endeavors. For example, the idea of patterns has been applied to describe data models [19], system level analysis and modeling information [16], software process and organizational structures [5], and curricula for educating software developers [22]. It is our intention in this paper to adapt and apply the notion of patterns to the description of specification of properties for finite-state verification.

3 What is a Specification Pattern?

A property specification pattern is a generalized description of a commonly occuring requirement on the permissible state/event sequences in a finite-state model of a system. A property specification pattern describes the essential structure of some aspect of a system's behavior and provides expressions of this behavior in a range of common formalisms.

Example specification patterns are given in Figures 1 and 2 (we use a variant of the "gang-of-four" pattern format [17]). A pattern comprises a name or names, a precise statement of the pattern's intent (i.e., the structure of the behavior described), mappings into common specification formalisms, examples of known uses, and relationships to other patterns.

Some specification formalisms (e.g., quantified regular expressions (QRE) [25]) are event-based, while others (e.g., various temporal logics, such as LTL and computation tree logic (CTL) [7]) are state-based. In our patterns, capital letters (e.g., P, Q, R, S) stand for events or disjunctions of events in event-based formalisms and stand for state formulas in state-based formalisms.

Each pattern has a *scope*, which is the extent of the program execution over which the pattern must hold. There are five basic kinds of scopes: global (the entire program execution), before (the execution up to a given state/event), after (the execution after a given state/event), between (any part of the execution from one given state/event to another given state/event) and after-until (like between but the designated part of the execution continues even if the second state/event does not occur). The scope is determined by specifying a starting and an ending state/event for the pattern: the scope consists of all states/events beginning with the starting state/event and up to but not including the ending state/event. Figure 3 illustrates the portions of an execution that are designated by the different kinds of scopes. We note that a scope itself should be interpreted as optional; if the scope delimiters are not present in an execution then the specification will be true.

Scope operators are not present in most specification formalisms (interval logics are an exception). Nevertheless, our experience strongly indicates that most informal requirements are specified as properties of program executions or segments of program executions. Thus a pattern system for properties should mirror this view to enhance usability.

Absence

Intent

To describe a portion of a system's execution that is free of certain events or states. Also known as **Never.**

Example Mappings

CTL P is false:

Globally	$AG(\neg P)$
Before R	$A[\neg P\,\mathcal{U}(R \vee AG(\neg R))]$
After Q	$AG(Q \rightarrow AG(\neg P))$
Between Q and R	$AG(Q \rightarrow A[\neg P\,\mathcal{U}(R \vee AG(\neg R))])$
After Q until R	$AG(Q \rightarrow \neg E[\neg R\,\mathcal{U}(P \wedge \neg R)])$

LTL P is false:

Globally	$\Box(\neg P)$
Before R	$\Diamond R \rightarrow \neg P\,\mathcal{U}\,R$
After Q	$\Box(Q \rightarrow \Box(\neg P))$
Between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow \neg P\,\mathcal{U}\,R)$
After Q until R	$\Box(Q \rightarrow \neg P\,\mathcal{U}(R \vee \Box \neg P))$

Quantified Regular Expressions Let Σ be the set of all events, let $[-P, Q, R]$ denote the expression that matches any symbol in Σ *except* P, Q, and R, and let $e^?$ denote zero or one instance of expression e.
Event P does not occur:

Globally	$[-P]^*$
Before R	$[-R]^* \mid [-P,R]^* R \Sigma^*$
After Q	$[-Q]^*(Q[-P]^*)^?$
Between Q and R	$([-Q]^* Q[-P,R]^* R)^*[-Q]^*(Q[-R]^*)^?$
After Q until R	$([-Q]^* Q[-P,R]^* R)^*[-Q]^*(Q[-P,R]^*)^?$

Examples and Known Uses

The most common example is mutual exclusion. In a state-based model, the scope would be global and P would be a state formula that is true if more than one process is in its critical section. For an event-based model, the scope would be a segment of the execution in which some process is in its critical section (i.e., between an enter section event and a leave section event), and P would be the event that some other process enters its critical section.

Relationships

This pattern is the dual of the **Existence** pattern. In fact, in many specification formalisms negation and explicit queries for existence will be used to formulate an instance of the **Absence** pattern, as seen in the examples above.

Figure 1: Absence Pattern

4 A System of Specification Patterns

We propose to develop a system of property specification patterns for finite-state verification tools. The pattern system is a set of patterns organized into one or more hierarchies, with connections between related patterns to facilitate browsing. A user would search for the appropriate pattern to match the requirement being specified, use the mapping section to obtain the essential structure of the pattern in the formalism used by a particular tool, and then instantiate that pattern by plugging in the state formulas or events specific to the requirement.

We believe the most useful way to organize the patterns will be in a hierar-

Response

Intent

To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as **Follows** and **Leads-to.**

Example Mappings

In these mappings P is the cause and S is the effect.

CTL S responds to P:

Globally	$AG(P \rightarrow AF(S))$
Before R	$A[(P \rightarrow A[\neg R\,\mathcal{U}(S \vee AG(\neg R))]) \,\mathcal{U}(R \vee AG(\neg R))]$
After Q	$AG(Q \rightarrow AG(P \rightarrow AF(S)))$
Between Q and R	$AG(Q \rightarrow A[(P \rightarrow A[\neg R\,\mathcal{U}(S \vee AG(\neg R))]) \,\mathcal{U}(R \vee AG(\neg R))])$
After Q until R	$AG(Q \rightarrow \neg E[\neg R\,\mathcal{U}(\neg P \rightarrow A[\neg R\,\mathcal{U}\,S]) \wedge \neg R])$

LTL S responds to P:

Globally	$\Box(P \rightarrow \Diamond S)$
Before R	$(P \rightarrow (\neg R\,\mathcal{U}\,S)) \,\mathcal{U}(R \vee \Box\neg R)$
After Q	$\Box(Q \rightarrow \Box(P \rightarrow \Diamond S))$
Between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow (P \rightarrow (\neg R\,\mathcal{U}\,S)) \,\mathcal{U}\,R)$
After Q until R	$\Box(Q \rightarrow ((P \rightarrow (\neg R\,\mathcal{U}\,S)) \,\mathcal{U}\,R) \vee \Box(P \rightarrow (\neg R\,\mathcal{U}\,S)))$

Quantified Regular Expressions Let Σ be the set of all events, let $[-P]$ denote the expression that matches any symbol in Σ *except* P, and let $e^?$ denote zero or one instance of expression e.
S responds to P:

Globally	$[-P]^*(P[-S]^*S[-P]^*)^*$
Before R	$[-R]^* \vert [-P,R]^*(P[-S,R]^*S[-P,R]^*)^*R\Sigma^*$
After Q	$[-Q]^*(Q[-P]^*(P[-S]^*S[-P]^*)^*)^?$
Between Q and R	$[-Q]^*(Q[-P,R]^*(P[-S,R]^*S[-P,R]^*)^*R[-Q]^*)^*$ $(Q[-R]^*)^?$
After Q until R	$[-Q]^*(Q[-P,R]^*(P[-S,R]^*S[-P,R]^*)^*R[-Q]^*)^*$ $(Q[-P,R]^*(P[-S,R]^*S[-P,R]^*)^*)^?$

Examples and Known Uses

Response properties occur quite commonly in specifications of concurrent systems. Perhaps the most common example is in describing a requirement that a resource must be granted after it is requested.

Relationships

Note that a **Response** property is like a converse of a **Precedence** property. **Precedence** says that some cause precedes each effect, and **Response** says that some effect follows each cause. They are not equivalent, because a **Response** allows effects to occur without causes (**Precedence** similarly allows causes to occur without subsequent effects).

Note that this pattern does not require that each occurrence of a cause will have its own occurrence of an effect.

Figure 2: Response Pattern

chy based on their semantics. For example, some patterns require states/events to occur or not occur (e.g., the Absence pattern), while other patterns constrain the order of states/events (e.g., the Response pattern). One organization for our pattern system is the hierarchy illustrated in Figure 4. This hierarchy distinguishes properties that deal with the occurrence and ordering of states/events

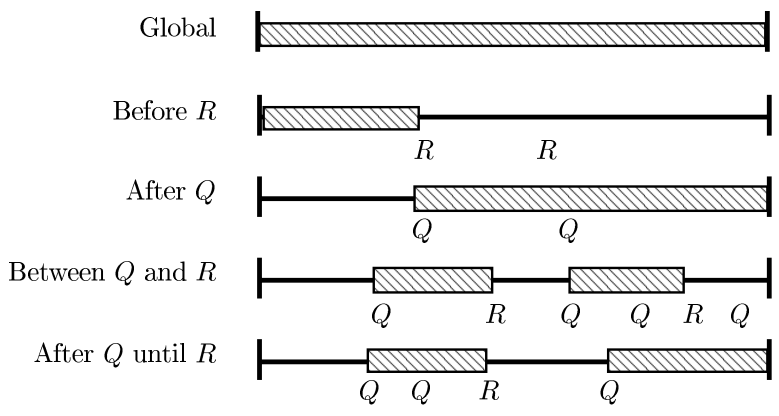

Figure 3: Pattern Scopes

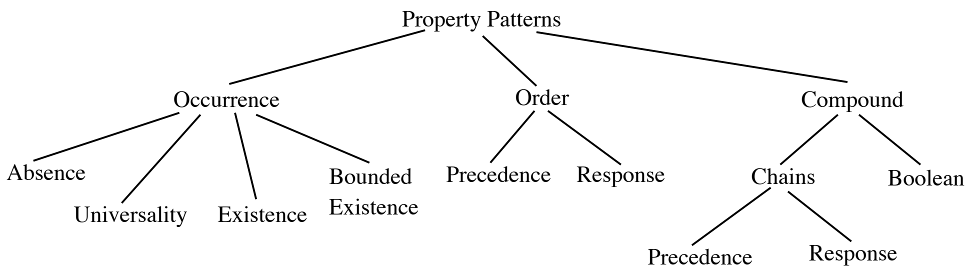

Figure 4: A Pattern Hierarchy

during system execution. It also provides separate patterns for compound properties that are built up from combinations of more basic patterns. Since different users may think about patterns in different ways, patterns could appear under several categories. For example, the Absence pattern with a non-global scope, in Figure 1, could be seen to constrain the order of states/events and could be put under ordering patterns in Figure 4. Patterns can also be organized into hierarchies based on their syntactic structure. This would allow someone who can specify a property in one formalism to find the corresponding pattern quickly, from which he or she could determine how to specify the property in another formalism. (Note that the hierarchy shown here does not explicitly address fairness issues, which will be included in the complete pattern system.)

In defining a specification formalism, one attempts to give a small set of independent concepts from which a large class of interesting specifications can be constructed. With the collection of specification patterns, however, we are neither trying to give a smallest set that can generate the useful specifications nor a complete listing of specifications. Patterns are in the system because they appear frequently as property specifications. In our experience, only a small fraction of the possible constraints that can be specified using logics or regular expressions commonly occur in practice.

4.1 The Patterns

Space limitations prohibit description of the patterns in full detail; for that we have set up a web-site [13]. The full patterns will contain additional examples, explanation of pattern relationships, and mappings in other formalisms including Graphical Interval Logic (GIL) [12], the INCA query language [11], automata, and various process algebra formalisms (e.g., CCS and CSP) [8]. Here we give the intent of some common patterns.

Occurrence Patterns include

Absence A given state/event does not occur within a scope. This pattern is also known as **Never**. Figure 1 gives the key elements of the pattern.

Existence A given state/event must occur within a scope. This pattern is also known as **Future** and **Eventuality**.

Bounded Existence A given state/event must occur k times within a scope. Variants of this pattern specify at least k occurrences and at most k occurrences of a state/event.

Universality A given state/event occurs throughout a scope. This pattern is also known as **Globally**, **Always** and **Henceforth**.

Ordering Patterns include

Precedence A state/event P must always be preceded by a state/event Q within a scope.

Response A state/event P must always be followed by a state/event Q within a scope. This pattern is also known as **Follows** and **Leads-to**. This pattern is a a mixture of **Existence** and **Precedence**, and expresses a causal relationship between two subject patterns. Figure 2 gives the key elements of the pattern.

Compound Patterns include

Chain Precedence A sequence of states/events $P_1, \ldots, P_n$ must always be preceded by a sequence of states/events $Q_1, \ldots, Q_m$. This pattern is a generalization of the **Precedence** pattern.

Chain Response A sequence of states/events $P_1, \ldots, P_n$ must always be followed by a sequence of states/events $Q_1, \ldots, Q_m$. This pattern is a generalization the **Response** pattern. It can be used to express bounded FIFO relationships.

Boolean Combinations Most of the patterns delimit scopes and describe inter-scope properties in terms of individual events/states. There are cases where we want to generalize the patterns to allow for sets of states/events to describe scopes and properties. In some cases this is straightforward and disjunctions or conjunctions of state/event descriptions can be substituted into patterns; in other cases this yields the incorrect specification. These patterns outline how boolean combinations can be applied in different cases.

4.2 Related Work

There have been some attempts at describing taxonomies for property specifications.

The most popular and long-lived of these distinguishes *safety* and *liveness* properties [20]. While this provides a very high-level intuitive understanding of classes of specifications, i.e., "nothing bad will ever happen" vs. "something good will eventually happen" it is much too coarse to be of practical use in constructing particular specifications.

Manna and Pnueli [21] describe a finer taxonomy based on the syntactic structure of LTL formulae. This taxonomy is defined in terms of canonical forms. Some of these forms do not match the way that specifications are typically encoded in LTL, so they provide some alternate codings for canonical forms. They also give number of examples along with textual descriptions of the intuition behind the specifications. This is the closest thing to a pattern catalog that appears in the literature on specification of concurrent and reactive systems. Unfortunately, even this taxonomy is too coarse for many practitioners. Furthermore, since it is syntactic in nature, it suffers from two additional drawbacks from the standpoint of practitioners. First, it specific to LTL, while practitioners using particular finite-state verification tools may need to couch their specifications in another formalism. Second, practitioners do not naturally attack a problem starting from its syntax in a particular specification formalism, but rather begin from an informal understanding of the meaning of the requirements. Thus, a taxonomy organized by features related to meaning is more appropriate.

5 Experience with a Small Pattern System

We have taught a one semester graduate course in specification and verification of reactive systems. This course is a component of a Masters of Software Engineering curriculum. The students in the course were almost exclusively non-traditional students working in the software industry. A number of these students had never taken a logic or discrete mathematics course while a few had taken such courses in the distant past. We believe this group is representative of a broad class of practicing software developers who are clearly not experts in formal methods. The course involved development of a significant collection of formal specifications for selected realistic systems including graphical user interfaces [14], transactional processing in an inventory control system, an automobile control system, and a home security system. These specifications were derived from informal English language statements of system requirements, which had been refined into a stylized structured English. The specifications were subsequently verified using model-checking.

We presented the writing and reading of formal specifications as a process of identifying and composing a small set of patterns similar to those described in Section 4. Students were able to write correct specifications for their projects in a matter of weeks; they also were able to read and critique each others specifications. While the project work focused mostly on CTL as a specification formalism, students were also required to express properties in other formalisms including LTL, QREs, finite state automata, and GIL. The benefits of the pattern-based approach stood out here since for most of these formalisms mapping from CTL to patterns and back to alternative formalisms is straightforward.

While our experience to date is limited and anecdotal we do feel that the pattern-based approach is useful as an educational tool and as a means of transferring expert knowledge. All students in the course, including those with the weakest formal backgrounds, were capable of producing readable specifications, in more than one specification language, by the end of the semester. Subsequently, it has become clear that at least three students have internalized the pattern system and mappings sufficiently well to apply them independently in projects for their master's degrees. We plan on using the pattern system again and gathering additional anecdotal evidence while teaching the course a second time in the Fall of 1997.

6 Conclusions

Patterns and the people who define them are characterized by an "aggressive disregard for originality" [4]. Patterns are not research; they are an expression of best-practice in a software domain. A pattern system does not belong to an individual, but to the community of experts and practitioners who contribute to and use it. It is important that a pattern system be agreed upon by that community. For these reasons, the system described in this paper should only be viewed as a starting point. If it is to become useful it must grow through a process of open dialog and critical review. There are efforts underway in other pattern domains (e.g., [9]) to provide a web-based mechanism for such collaborative development of pattern systems. We believe that specification patterns would flourish if a similar collaboration were undertaken by the formal specification community and towards this end we have set up a web-site [13] to store the current and future versions of the specification patterns system. In particular, we want to stress our belief that users of specification patterns will benefit from a variety of views of the pattern system, reflecting semantic hierarchies, the syntactic structure of specifications in particular formalisms, and other organizing principles, and we hope the web site will be a vehicle for collecting and disseminating these views as well as individual patterns.

In this paper, we have suggested that finite-state verification might be made more readily accessible to developers of concurrent and reactive systems through definition and use of a pattern system. We have describe an initial version of a pattern system for specification of properties for finite-state verification tools. Our experience suggests that such a pattern system enables non-experts to become proficient at writing and reading formal specifications for realistic systems relatively quickly.

Development of a pattern system is a community activity requiring participation by a broad range of experts both in patterns and in the formal specification domain. It is our hope that such a collaboration will become a reality and that the resulting pattern system will be studied and put to use by practicing developers.

References

[1] R. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *Software Engineering Notes*, 21(6):156–166, Nov. 1996. Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[2] B. Appleton. Patterns and Software: Essential Concepts and Terminology. `http://www.enteract.com/~bradapp/docs/patterns-intro.html`, 1997.

[3] G. Avrunin, U. Buy, J. Corbett, L. Dillon, and J. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, Nov. 1991.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons Ltd., 1996.

[5] B. G. Cain, J. O. Coplien, and N. B. Harrison. Social Patterns in Productive Software Organizations. *Annals of Software Engineering*, 2:259–286, Dec. 1996.

[6] A. Chamillard. *An Empirical Comparison of Static Concurrency Analysis Techniques*. PhD thesis, University of Massachusetts at Amherst, May 1996.

[7] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986.

[8] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, Jan. 1993.

[9] J. Coplien. Organizational Patterns. http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?Organizationa%lPatterns, 1997.

[10] J. Coplien and D. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[11] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, Jan. 1995.

[12] L. K. Dillon, G. Kutty, L. 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, Apr. 1994.

[13] M. Dwyer, G. Avrunin, and J. Corbett. A System of Specification Patterns. http://www.cis.ksu.edu/~dwyer/spec-patterns.html, 1997.

[14] M. 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*, Sept. 1997. to appear.

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

[16] M. Fowler, editor. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[18] Z. Har'El and R. P. Kurshan. Software for analytical devleopment of communication protocols. *AT&T Technical Journal*, 69(1):44–59, 1990.

[19] D. Hay, editor. *Data Model Patterns: Conventions of Thought*. Dorset House Publishing, 1995.

[20] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[21] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[22] M. Manns, H. Sharp, P. McLaughlin, and M. Prieto. Pedagogical Patterns: Successes in Teaching Object Technology. http://www.cs.unca.edu/~manns/oopsla.html, 1997.

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

[24] G. Naumovich, L. Clarke, and L. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Oct. 1996.

[25] K. Olender and L. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, Mar. 1990.

[26] D. Rosenblum. Formal methods and testing: Why the state-of-the-art is not the state-of-the-practice (ISSTA'96/FMSP'96 panel summary). *ACM SIGSOFT Software Engineering Notes*, 21(4), July 1996.

[27] J. Wing and M. Vaziri-Farahani. Model checking software systems: A case study. *Software Engineering Notes*, 20(4):128–139, Oct. 1995. Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering.