Patterns in Property Specifications for Finite-State Verification*

Matthew B. Dwyer
Kansas State University
Department of Computing and Information Sciences
Manhattan, KS 66506-2302
+1 785 532 6350
dwyer@cis.ksu.edu

George S. Avrunin
University of Massachusetts
Department of Mathematics and Statistics
Amherst, MA 01003-4515
+1 413 545 4251
avrunin@math.umass.edu

James C. Corbett
University of Hawai'i
Department of Information and Computer Science
Honolulu, HI 96822
+1 808 956 6107
corbett@hawaii.edu

ABSTRACT

Model checkers and other finite-state verification tools allow developers to detect certain kinds of errors automatically. Nevertheless, the transition of this technology from research to practice has been slow. While there are a number of potential causes for reluctance to adopt such formal methods, we believe that a primary cause is that practitioners are unfamiliar with specification processes, notations, and strategies. In a recent paper, we proposed a pattern-based approach to the presentation, codification and reuse of property specifications for finite-state verification. Since then, we have carried out a survey of available specifications, collecting over 500 examples of property specifications. We found that most are instances of our proposed patterns. Furthermore, we have updated our pattern system to accommodate new patterns and variations of existing patterns encountered in this survey. This paper reports the results of the survey and the current status of our pattern system.

Keywords

Patterns, finite-state verification, formal specification, concurrent systems

1 INTRODUCTION

Although formal specification and verification methods offer practitioners some significant advantages over the current state-of-the-practice, they have not been widely adopted. Partly this is due to a lack of definitive evidence in support of the cost-saving benefits of formal methods, but a number of more pragmatic barriers to adoption of formal methods have been identified [22], including the lack of such things as good tool support, appropriate expertise, 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 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 [19], bisimulation [4], language containment [14], flow analysis [10], and inequality necessary conditions [1]. In contrast to mechanical theorem proving, which often requires guidance by an expert, most finite-state verification techniques can be fully automated, relieving the user of the need to understand the inner workings of the verification process. Finite-state verification techniques are especially critical in the development of concurrent systems, where non-deterministic behavior makes testing especially problematic.

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. This is more challenging than it might at first appear. For example, consider the following requirement for an elevator: *Between the time an elevator is* call*ed at a floor and the time it* open*s its doors at that floor, the elevator can arrive* at that floor *at most twice.* To verify this property with a linear temporal logic (LTL) model checker, a developer would have to translate this informal requirement into the following LTL formula:

$$\begin{array}{l}\Box((\text{call} \wedge \Diamond\text{open}) \rightarrow \\ \quad((\neg\text{atfloor} \wedge \neg\text{open})\ \mathcal{U} \\ \quad\quad(\text{open} \vee ((\text{atfloor} \wedge \neg\text{open})\ \mathcal{U} \\ \quad\quad\quad(\text{open} \vee ((\neg\text{atfloor} \wedge \neg\text{open})\ \mathcal{U} \\ \quad\quad\quad\quad(\text{open} \vee ((\text{atfloor} \wedge \neg\text{open})\ \mathcal{U} \\ \quad\quad\quad\quad\quad(\text{open} \vee (\neg\text{atfloor}\ \mathcal{U}\ \text{open})))))))))\end{array}$$

Not only is this formula difficult to read and understand, it is even more difficult to write correctly without some expertise in the idioms of the specification language.

*This work was partially supported by NSF grants CCR-9407182, CCR-9633388, CCR-9703094, and CCR-9708184 and by NASA grant NAG-02-1209.

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. Even with significant expertise, dealing with the complexity of such a specification can be daunting. In many software development phases, such as design and coding, complexity is addressed by the definition and use of abstractions. For complex specification problems, abstraction is just as important.

In [9], we proposed to capture the experience base of expert specifiers and enable the transfer of that experience between practitioners by way of a *specification pattern system*. This system is, essentially, a collection of parameterizable, high-level, formalism-independent specification abstractions. To maximize the coverage of these abstractions, we have described a variety of techniques for tuning their semantics to meet the needs of different users. We adopted a pattern-based approach to presenting our specification abstractions because of its focus on the matching of problem characteristics to solution strategies. Patterns were originally developed to capture recurring solutions to design and coding problems [12]. 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 commonly occuring problems.

We hypothesized that a pattern-based approach would also be successful in the domain of property specifications for finite-state verification. While there are a number of very expressive formalisms, such as CTL*, most of the specifications that we knew about fell into a relatively small number of categories. Thus, we believed that a collection of simple patterns could be defined to assist practitioners in mapping descriptions of system behavior into their formalism of choice, and that this might improve the transition of these formal methods to practice.

To evaluate our hypothesis, we surveyed all the sources of property specifications we could locate and collected over 500 examples of property specifications for finite-state verification tools. As expected, we found that the vast majority (92%) are instances of patterns in our system. We subsequently updated the pattern system to accommodate new patterns and variations of existing patterns encountered in the survey. This paper gives an overview of our updated pattern system and reports the results of our survey of property specifications, the only study of its kind we are aware of.

Section 2 describes our specification pattern system for finite-state verification. Section 3 presents the results of our survey. Section 4 compares our approach with related work and Section 5 concludes.

2 A SPECIFICATION PATTERN SYSTEM

In this section, we describe our pattern system. We begin by giving some background on the notion of patterns. We then describe how this idea can be applied to the domain of property specifications for finite-state verification. Finally, we describe how a set of such patterns can be organized and we give an overview of the current state of our pattern system.

Property Specification Patterns

Design patterns were introduced [12] 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.

For finite-state verification, the system is modeled as a transition system with a finite number of states and a set of transitions, possibly labeled with events, between these states. A property specification pattern is a generalized description of a commonly occurring requirement on the permissible state/event sequences in such 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.

An example of a property specification pattern is given in Figure 1 (we use a variant of the "gang-of-four" pattern format [12]). 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) [20]) are event-based, while others (e.g., various temporal logics, such as LTL and computation tree logic (CTL) [3]) 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.

Precedence

Intent

To describe a relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first. Also known as *Enables*.

Example Mappings

In these mappings S enables the occurrence of P.

CTL S precedes P:

Globally	$\neg E[\neg S\ \mathcal{U}(P \wedge \neg S))]$
Before R	$\neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R \wedge EF(R))]$
After Q	$\neg E[\neg Q\ \mathcal{U}(Q \wedge \neg E[\neg S\ \mathcal{U}(P \wedge \neg S)])]$
Between Q and R	$AG(Q \to \neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R \wedge EF(R))])$
After Q until R	$AG(Q \to \neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R)])$

LTL S precedes P:

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

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 S precedes P:

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

Examples and Known Uses

Precedence properties occur quite commonly in specifications of concurrent systems. One example is describing a requirement that a resource (e.g., a lock) is only granted in response to a request.

Precedence and response properties often go together. A response property says that when S occurs then an occurrence of P must follow. If we want to restrict P to only follow S then we use a precedence property. Note that these properties do not guarantee a one-to-one correspondance between an occurrence of S and an occurrence of P. Such additional constraints can be added using the *constrained* variations of these patterns.

The mappings given in this pattern do not describe precedence properties where P and S occur simultanously (i.e., S must strictly precede P). To relax this constraint use the *possibly empty* variation of the pattern.

Relationships

A generalization of precedence properties that allows for multiple separate states/events to constitute P and S is called the precedence chain pattern.

Figure 1: Precedence Pattern

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.

For state-delimited scopes, the interval in which the property is evaluated is closed at the left and open at the right end. Thus, the scope consists of all states beginning with the starting state and up to but not including the ending state. We chose closed-left open-right scopes

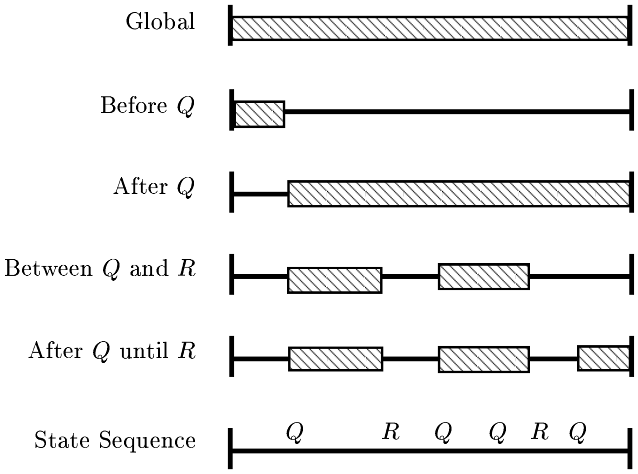

Figure 2: Pattern Scopes

because they are relatively easy to encode in specifications and they have been the most commonly encountered in the real property specifications we studied. It is possible, however, to define scopes that are open-left and closed-right; we explain how to construct these variants of the mappings in a special part of the pattern system (described below). In event-based formalisms the underlying model does not allow two events to coincide, thus event-delimited scopes are open at both ends.

Figure 2 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 indicates that many informal requirements are specified as properties of segments of program executions. Thus a pattern system for properties should mirror this view to enhance usability.

We note that the various specification formalisms have different semantics and expressive power, and that a property that can be expressed easily in one formalism may be unnatural, or even impossible to capture precisely, in a different formalism. For instance, in state-based formalisms such as LTL or CTL, it is reasonable to specify that a certain proposition hold throughout a scope (the Universality pattern), and to regard this as being in some sense dual to the Absence property stating that a proposition holds at no state in the scope. In event-based formalisms, although it is easy to require that only certain events occur within a scope, the property that a proposition holds throughout the scope would probably be expressed in terms of the appropriate occurrence of an event indicating that the proposition has become true and the absence of an event indicating that it has become false, which does not bear a simple relation to the Absence pattern. Similarly, we note that some formalisms can express conditions involving infinite executions, while others are limited to finite sequences of states or events. Although we expect that, in practice, almost all of the properties to be specified can be expressed in almost all of the commonly used formalisms, the pattern system should point out these differences to the user.

A System of Specification Patterns

We have developed 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 a template of the property in the formalism used by a particular tool, and then instantiate that template by plugging in the state formulas or events specific to the requirement.

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. We hypothesize that only a small fraction of the possible properties that can be specified using logics or regular expressions commonly occur in practice. These properties, and simple variants of them, make up our pattern system. We expect the set of patterns to grow over time as developers encounter property specifications of real systems that do not easily map onto the existing patterns.

The Patterns

Space limitations prohibit description of the patterns in full detail; for that we have set up a web-site [8]. The full patterns contain additional examples, explanation of relationships among the patterns, and mappings to various formalisms. A list of our set of patterns, with short descriptions, follows. In the descriptions, for brevity, we use the phrase "a given state/event occurs" to mean "a state in which the given state formula is true, or an event from the given disjunction of events, occurs."

Absence A given state/event does not occur within a scope.

Existence A given state/event must occur within a scope.

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. The elevator property in Section 1 is an instance of this pattern.

Universality A given state/event occurs throughout a scope.

Precedence A state/event P must always be preceded by a state/event Q within a scope. Figure 1 gives the key elements of the pattern.

Response A state/event P must always be followed by a state/event Q within a scope.

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.

Organization
We believe the most useful way to organize the patterns is in a hierarchy 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 3. This hierarchy distinguishes properties that deal with the occurrence and ordering of states/events during system execution.

In addition to the patterns themselves, we provide a set of *pattern notes*, which explain how to combine and/or vary the patterns. For example, pattern templates are typically parameterized by individual events or state formulae. In some cases, however, we can allow patterns of states/events to be substituted into the templates. Pattern notes give the user guidance on when it is safe to make these substitutions. There are also pattern notes describing how to construct known variants of the patterns, such as those with left-open right-closed scopes, or those in which specific states/events are absent from segments of a pattern.

In the original pattern system described in [9], we provided mappings to three formalisms: LTL, CTL, and QREs. We have since added mappings for two additional formalisms: Graphical Interval Logic (GIL) [7] and the INCA query language [5] (due to space limitations, we do not show these mappings in Figure 1).

3 SURVEY OF PROPERTY SPECIFICATIONS

In [9], we first proposed the idea of a pattern system for property specifications for finite-state verification. We assumed that the specifications people write fall into a small number of categories, although we gave little empirical evidence for this important assumption. Since then, we have collected over 500 example specifications and found that, indeed, most fall into a small number of familiar categories. We describe our survey of property specifications in this section.

Data Collection
We collected example specifications from:

- Verification papers in the literature.
- Others who have written/collected specifications. In particular, we contacted researchers who have built/used verification tools and asked for example specifications. We also sent a request for examples to several mailing lists and newsgroups.
- Student projects from two offerings of the first author's graduate course in finite-state verification.

In all, we collected 555 specifications from at least 35 different sources. The specifications collected were in many forms. For most we had an expression of the requirement in a specific specification formalism (e.g., LTL). For many we also had an informal prose description of the requirement. The specifications came from a wide variety of application domains, including: hardware protocols, communication protocols, GUIs, control systems, abstract data types, avionics, operating systems, distibuted object systems, and databases. Complete descriptions of these specifications can be found on our Specification Patterns web page [8], along with citations for 34 published papers from which many of them were taken.

We examined each specification and manually determined whether it matched a pattern and, if so, the scope of the property. In most cases, the formal version of the specification was an instantiation of a template mapping for a specific pattern/scope; in this case the classification was trivial. If we could not find a trivial match, we looked at the specification more carefully and still counted it as a match if:

- *The specification was formally equivalent to an instantiation of one of our template mappings.* For example, the LTL formula $\neg\Diamond P$ is equivalent to $\Box\neg P$, which is our mapping for (global) absence of P.
- *The specification can be obtained from one of our patterns using parameter substitution.* As described in Section 2, the template mapping parameters for logics are usually state-formulae (i.e.,

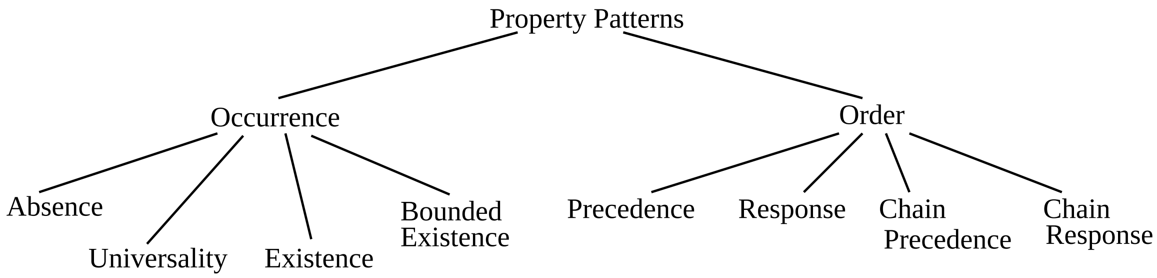

Figure 3: A Pattern Hierarchy

boolean combinations of propositions). For some patterns/scopes, it is safe to substitute temporal formulae into the template; guidance on when this is correct is given in a pattern note. We encountered 13 cases in which the requirement "infinitely often P" was realized using the CTL formula $AG(AF(P))$, which is a universal pattern instantiated with the parameter $AF(P)$.

- *The specification was a known variant of one of our patterns.* As noted in Section 2, the mappings for our scopes are closed on the left and open on the right. Nearly all of the non-global specifications we collected were left-closed right-open, but three were left-open right-closed. As mentioned in Section 2, a pattern note explains how to open the left end of the scope and how to close the right end. Other variants mentioned in the pattern notes include chain patterns where certain states or events are forbidden between elements of the chains.
- *The specification was a new variant of one of our patterns.* We discovered two interesting variants of the *Response* pattern. The first describes an execution in which S must respond to P *and* there must be no Zs between the (stimulus) P and the (response) S. The second requires that the response be in the next state/event. We found 41 instances of the constrained response variant and 8 instances of the next response variant. We have added pattern notes explaining how to adjust the mappings to obtain these variants.
- *The formal expression of the specification was clearly in error, and the correct specification was a instance of one of our patterns.* About 10 of the specifications we collected were clearly in error—either the formal expression did not match the prose description, or the formal expression did not make sense. When the intent of the specification was clear, and the intent matched one of the patterns, we counted the example as a match.

For each specification, we recorded the following information (when available):

Requirement A prose description of the requirement.

Pattern The pattern of which we determined this requirement is an instance (if any).

Scope The scope of the pattern.

Parameters Notes on the parameters provided to the template (e.g., arrays of propositions, nested temporal formulae).

Mappings Mappings of the property to formal specification languages (LTL, CTL, QREs, GIL, INCA). Most examples collected have exactly one mapping.

Source The source of the example (i.e., person, citation).

Domain The application domain the example is from.

Note Any additional information on the example.

An sample entry for a specification is:

```
REQUIREMENT: When a server requests its registration in
             the ROT, it will eventually be registered.
PATTERN: Response
SCOPE: Global
PARAMETERS: Propositions (boolean vector)
LTL: [](RequestedRegisterImpl[i] -> <>ServerRegistered[i])
NOTE: this is replicated for all modeled servers
SOURCE: Gregory Duval \cite{duval:98}, RI, pp. 46
DOMAIN: Distributed Object System
```

As noted above, all of the specifications we found are available on the World Wide Web at [8].

Results

The data are summarized in Table 1, which gives totals for each pattern/scope combination, and in Figure 4, which graphs the totals for each pattern and scope (examples not matching any pattern are grouped under UNKNOWN). Of the 555 example specifications we collected, 511 (92%) matched one of our patterns. As shown in Figure 4, the most common pattern in the sample is *Response*, with the next most common being *Universality* and its dual *Absence*. Together, these three patterns accounted for 80% of the sample.

Although we found at least one instance of each pattern, Figure 4 shows that a few patterns cover a majority of the sample. In fact, the frequencies of the patterns,

Pattern	Scope Glbl	Bfr	Aftr	Btwn	Untl	Tot
Absence	41	5	12	18	9	85
Universality	110	1	5	2	1	119
Existence	12	1	4	8	1	26
Bound Exist	0	0	0	1	0	1
Response	241	1	3	0	0	245
Precedence	25	0	1	0	0	26
Resp. Chain	8	0	0	0	0	8
Prec. Chain	1	0	0	0	0	1
UNKNOWN						44
Total	438	8	25	29	11	555

Table 1: Totals for Patterns/Scopes (All Data)

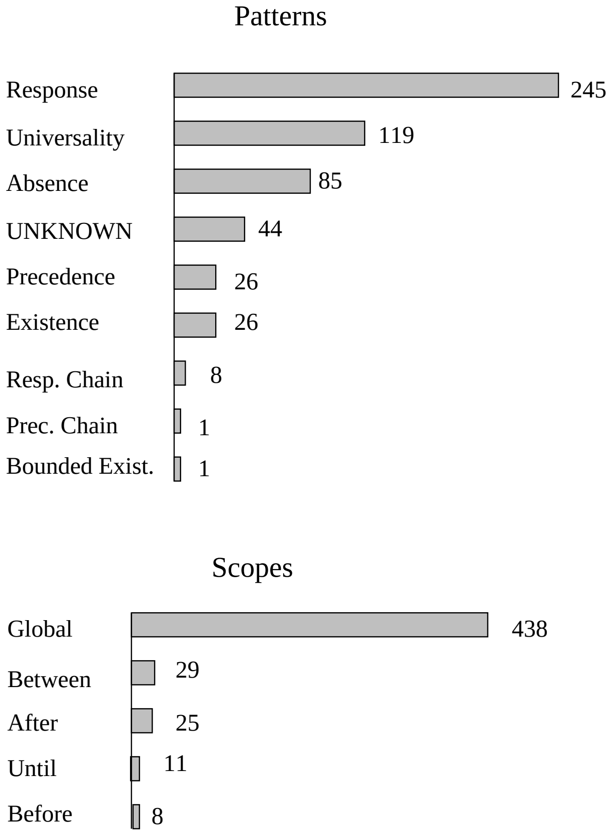

Figure 4: Total Number of Instances for Patterns/Scopes

when put in order, drop off very quickly. This raises the issue of size: how many patterns should be in the pattern system? The more patterns in the system, the more likely a match will be found. On the other hand, the system should be small enough to be easily browsed. Adding patterns that match very few real specifications may not be worth the slight increase in the pattern system's size.

As can be seen from Figure 4, most examples (80%) used a global scope. Also, note that almost all exam-

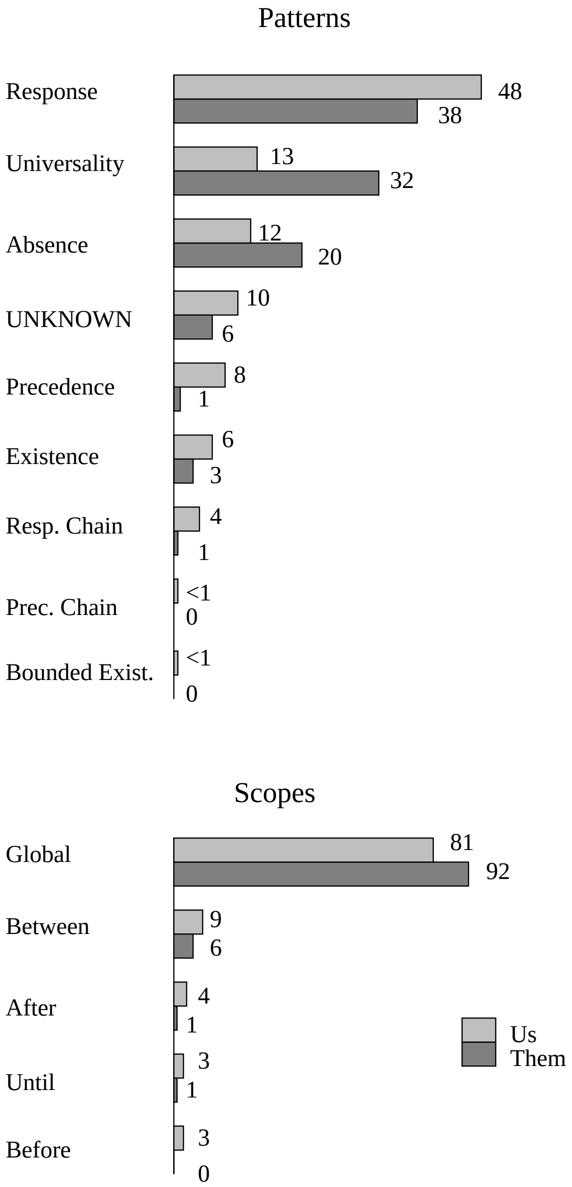

Figure 5: Percentage of Specifications for Each Pattern/Scope (Us vs. Them)

ples that used a non-global scope were instances of *Absence*, *Universality*, or *Existence*. These three patterns are conceptually the simplest; for global scopes, they map to single operators of temporal logics.

As mentioned above, some of the example specifications were collected from researchers and students who have a connection with one or more of us. To see whether the 304 specifications written by "us" (including our collaborators and students) differ significantly from the 251 specifications written by "them" (everyone else), we separate the data and compare the two data sets in Figure 5. Although there were some differences between the two data sets, note that the ranking of the

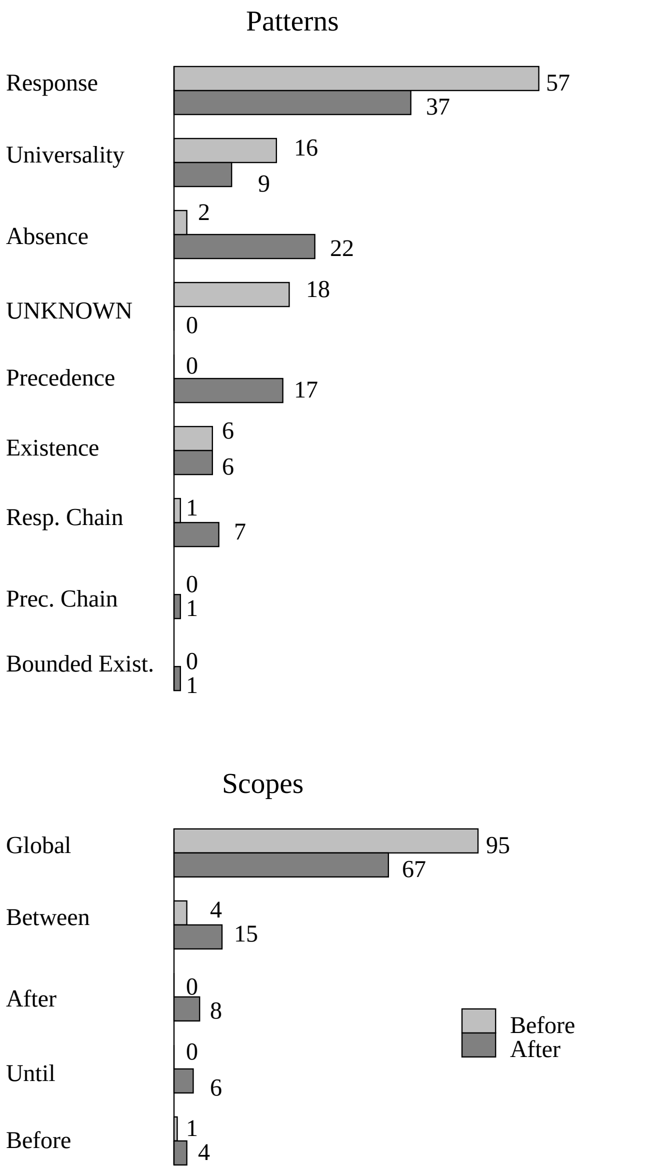

Figure 6: Percentage of Specifications for Each Pattern/Scope (Before vs. After)

four most common patterns (*Response, Universality, Absence, UNKNOWN*) is the same. The rankings for the remaining patterns were different, but given the small number of instances of these patterns in the data set (fewer than 10), the data are likely to be noisy. One difference is that we wrote more specifications with non-global scopes, although the vast majority of our specifications also used a global scope. Interestingly, we wrote more specifications that fall outside the pattern system than others.

Another interesting question is whether having the pattern system affects the kind of specifications people write. Finite-state verification is often used to check critical requirements, but the exact choice of requirements to be checked, as well as the way these requirements are expressed in terms of the model, is usually left to the analyst. Given a pattern system to assist in the formulation of property mappings, an analyst might be more likely to use more complex patterns and/or scopes. To investigate this, we took the set of specifications we or those affiliated with us had written (the "us" set from above), and divided it into the 165 specifications written before we developed the pattern system and the 139 specifications written after. This comparison is shown in Figure 6.

The differences in this case are more pronounced (although the sample size is smaller, so there may be more noise). Several more complex patterns (e.g., chains) only appear after the pattern system was introduced. More specifications created after the development of the pattern system used non-global scopes (33% vs. 5%), and those later specifications were more evenly (though not uniformly) spread over the different patterns. Also, note that all specifications written after the pattern system matched one of the patterns.

We have demonstrated that a pattern system for property specifications for finite-state verification is potentially useful by showing that most specifications fall into a small number of categories. To show that the system is actually useful would require answering several more difficult questions, including:

- Do patterns facilitate the learning of the specification formalisms?
- Do patterns allow specifications to be written more quickly?
- Are the specifications generated from patterns more likely to be correct?

Although we believe that the answer to all of these questions is yes, definitive answers would require experiments with human subjects. To date, we have only annecdotal evidence to support our claims, largely from the first author's experience teaching a finite-state verification course in a Masters level software engineering curriculum.

4 RELATED WORK

Traditionally, specification formalisms are presented in terms of a minimal set of operators in order to simplify their semantics definition. Additional operators are then defined in terms of those operators, e.g., $\Box P = \neg\Diamond\neg P$, so only a few primitives need to be formally defined. Users of such formalisms would prefer to write specifications at a higher level than this, so many formalisms provide built-in higher-level operators or abstractions. The classic example is *leads-to*, introduced by Owicki and Lamport [21], which plays a critical role in formalisms such as UNITY [2] and TLA [15]. Leads-to corresponds to a global response pattern in our system. A number of formalisms have been designed to allow for the definition of additional higher-level derived operators. For example, LUSTRE [13] property specifications, which are safety properties written in a linear-time temporal logic, are often written in terms of derived operators. In [13], the authors illustrate this capability by defining *since* and *never* operators in terms of the basic LUSTRE operators. These derived operators correspond to an existence with an after scope and an absence with a global scope, respectively. Our pattern system supports a common, but fixed set of specification abstractions in formalisms that lack explicit support for defining abstractions.

Like our patterns, interval logics (e.g., [7]) provide the user with a higher-level mechanism for defining the segments of the system's execution over which a property must hold. In fact, the interval construction operators provided by such logics are much more general than our five scopes. We have essentially taken five commonly occuring intervals and packaged them with the patterns.

Some recent work in requirements engineering has explored the use of templates or patterns in the construction of requirements. For instance, van Lamsweerde and his co-authors [6,18] have suggested using a library of refinements to construct detailed requirements from goals, and the Attempto Controlled English project [11], which uses a restricted subset of natural language to formulate requirements, offers annotated templates to guide non-expert users. These efforts are aimed at the development of essentially complete requirements for a system, while our pattern system is concerned more with the translation of particular aspects of such requirements into the formal specifications suitable for use with finite-state verification tools.

There has been little study of the classes of specifications that developers may or do write. Manna and Pnueli [17] address this issue from a theoretical angle by proposing a syntactic classification of LTL formula that completely describes the space of possible specifications one may write in LTL. Their taxonomy of specifications includes categories that are much broader than our patterns. As a consequence, most of our patterns are easily categorized in their classification: precedence, absence, and universal patterns with global scopes are all *safety* properties, existence patterns with global scope are *guarantee* properties, and response patterns with global scope are *response* properties. Our k-bounded existence pattern is very similar to their k-bounded overtaking property (which is a safety property). With chain patterns and complex scopes the mapping to syntactic categories is more difficult (due to the nature of the canonical forms which define the categories). In contrast to this work, the goal of our work is to give a constructive description of the classes of specifications that occur most frequently in practice.

In [16], Manna and Pnueli adopt a more pragmatic approach that is in line with the intent of our pattern system. They claim that very little of the general theory of temporal logic is required to handle the most important, and common, correctness properties of concurrent programs. They define a restricted proof-system that handles *invariance* properties (which subsume most universal and absence patterns), *response* properties (which subsume response patterns) and *precedence* properties (which subsume precedence and bounded existence and are similar to chain patterns). Their precedence properties also subsume after-until scope versions of universal and absence patterns. The data we present in this paper support Manna and Pnueli's intuition about common properties, since we found that absence, universal and response patterns constitute the bulk of the properties in our survey.

5 CONCLUSIONS

We believe that the definition and use of high-level abstractions in writing formal specifications is an important factor in making automated formal methods, specifically finite-state verification tools, more usable. Our specification pattern system provides a set of commonly occuring high-level specification abstractions for formalisms that do not support the definition of such abstractions directly. We have described an updated pattern system we developed for property specifications in finite-state verification and have collected a large sample of specifications that suggests that most property specifications people write are instances of patterns in this system.

We are currently exploring several directions for further work on specification patterns. We are working to define patterns to simplify writing a class of assume-guarantee properties in CTL. Writing such specifications in LTL is straightforward; in CTL, however, such properties can be quite tricky to express. We are studying the benefits of defining a language for property specification based on the patterns, providing automated support for compiling properties expressed in that language to specific formalisms and checking the legality of pattern substitutions. We are studying approaches for checking the consistency of pattern mappings expressed in multiple formalisms. For example, while CTL and LTL are not, in general, co-expressive, all of our CTL and LTL mappings lie in the all-paths fragment of CTL*; thus we may be able to formally justify the equivalence of those mappings.

The survey described in this paper is certainly not exhaustive, and we expect that, as the application of finite-state verification technology spreads, the types of specifications used by developers may change over time. We view the pattern system as a dynamic entity that will grow through a process of dialog and critical review by the community of developers and users of finite-state verification techniques and we welcome contributions from that community.

ACKNOWLEDGMENTS

We would like to thank Laura Dillon, Hamid Alavi and Corina Pasareanu for making significant contributions to the pattern system. We would like to thank the following people for contributing specifications, directly or indirectly: G. Holzmann, B. Yang, T. Chamillard, F. Cassez, A. van Lamsweerde, E.H. Ache, J. Atlee, S. Berezin, A. Biere, V. Carr, T. Cattel, W. Chan, A. Cimatti, E. Clarke, R. Cleaveland, B. Donahue, G. Duval, Y. Gao, F. Giunchiglia, R. Gulla, K. Havelund, L. Hines, J. Isom, H. Ejersbo Jensen, J. Julliand, K.G. Larsen, P.B. Ladkin, S. Leue, M. Lowry, P. Merino, J. McClelland, T. McCune, G. Mongardi, T. Nakatani, G. Naumovich, M. Novak, J. Penix, S. Probst, M. Ryan, A. Skou, A. Srikanth, F. Torielli, P. Traverso, J. M. Troya, M. Vaziri-Farahani, J.M. Wing, and K. Winter.

REFERENCES

[1] 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.

[2] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1988.

[3] 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.

[4] 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.

[5] 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.

[6] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In D. Garlan, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–190, San Francisco, Oct. 1996. ACM (Proceedings appeared in *Software Engineering Notes*, 21(6).

[7] 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.

[8] M. Dwyer, G. Avrunin, and J. Corbett. A System of Specification Patterns. `http://www.cis.ksu.edu/santos/spec-patterns`, 1997.

[9] M. Dwyer, G. Avrunin, and J. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, Mar. 1998.

[10] 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 Second ACM SIGSOFT Symposium on Foundations of Software Engineering.

[11] N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). In *CLAW 96, the First International Workshop on Controlled Language Applications*, 1996.

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

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous programming language LUSTRE. *Proceedings of the IEEE*, 79(9), Sept. 1991.

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

[15] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[16] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical Report STAN-CS-90-1321, Stanford University, July 1990. appeared in *Carnegie Mellon Computer Science: A 25 year Commemorative*, ACM Press, 1990.

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

[18] P. Massonet and A. van Lamsweerde. Analogical reuse of requirements frameworks. In *Proceedings of RE '97—Third International Conference on Requirements Engineering*, 1997.

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

[20] 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.

[21] S. S. Owicki and L. Lamport. Proving liveness properties of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 4:455–495, July 1982.

[22] 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.