PROPEL: An Approach Supporting Property Elucidation

Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003
(413) 545-2013
{rasmith, avrunin, clarke, ljo}@cs.umass.edu

ABSTRACT

Property specifications concisely describe what a software system is supposed to do. It is surprisingly difficult to write these properties correctly. There are rigorous mathematical formalisms for representing properties, but these are often difficult to use. No matter what notation is used, however, there are often subtle, but important, details that need to be considered. PROPEL aims to make the job of writing and understanding properties easier by providing templates that explicitly capture these details as options for commonly-occurring property patterns. These templates are represented using both "disciplined" natural language and finite-state automata, allowing the specifier to easily move between these two representations.

1. INTRODUCTION

Finite-state verification approaches, such as model checking, determine if the behavior of a hardware or software system is consistent with a specified property. Instead of specifying the full behavior of the system, each property may focus on one particular aspect of system behavior. These properties may be written in a number of different specification formalisms, such as temporal logics, graphical finite-state machines, or regular expression notations, depending on the finite-state verification system that is being employed. Although there are sometimes theoretical differences in the expressive power of these languages, these differences are rarely encountered in practice. A serious problem that is frequently encountered in practice, however, is expressing the intended behavior of the system correctly. Even though properties usually focus on some restricted aspect of a system's behavior, it is still surprisingly difficult to capture this behavior precisely. These properties are often "almost" correct, but fail to capture some important, and sometimes subtle, aspects of the system's intended behavior. Often these aspects are not revealed until testing or verification. Thus, analysts frequently spend a considerable amount of time trying to verify a property, only to later determine that the property has been specified incorrectly.

Software developers tend to avoid the more mathematical property specification formalisms and instead write requirements and design specification documents in natural language, perhaps sprinkled with some tabular or graphical notations. Although these documents seem to be more accessible to practitioners, they are usually very verbose and contain imprecise—and sometimes ambiguous and inconsistent—descriptions of the system. Thus, they are of limited value when doing rigorous analysis of the system.

What is needed is a property specification approach that is not only accessible to developers, but is also mathematically precise, so that it can be used as the basis for verification and other types of analysis. Recent work on property patterns [8-10] recognized that the properties used in formal verification often map onto one of several basic property patterns. These patterns can be instantiated with specific events or states and then mapped to several different formalisms. When we tried to employ these property patterns to represent some actual natural language requirements, however, we found that they were not adequate. They failed to represent some of the subtle differences in interpretation that we encountered.

In the work presented here, we build upon the property patterns in several important ways. First, we extend the patterns so that they are represented by pattern templates. Thus, instead of just parameterizing the pattern in terms of the events or states, we extend the patterns with alternative options that are explicitly shown to the specifier. Choosing among these options should help the specifier consider the relevant alternatives and subtleties associated with the intended behavior. Second, we represent these pattern templates using two different notations: an extended finite-state automaton (FSA) representation and a disciplined natural language (DNL) representation. Both of these representations have some advantages. The DNL representation provides a short list of alternative phrases that highlight the options, as well as synonyms for each option to support customization. This representation should appeal to those specifiers who prefer a natural language description. The extended FSA representation provides a graphical view that can be used to derive an instantiation of a specific FSA representation. It too helps the specifier see the options that need to be considered. Third, the instantiated FSA representation is mathematically well-defined and thus can be used as the basis for verification, as well as for testing the acceptance of event sequences, validating the consistency of a set of property automata, or other types of analysis. Finally, we believe that providing specifiers with the ability to view both representations simultaneously and select the available options from either representation will help them to elucidate the desired property. We are currently developing a

ICSE'02, May 19-25, 2002, Orlando, Florida, USA.

system, called PROPEL, for "PROPerty ELucidation," that provides support for specifying properties based on the property pattern templates, using these two complementary representations.

This paper describes property pattern templates and the PROPEL system. The next section of the paper reviews property patterns and explains the concerns that motivated the extension to the templates. Section 3 describes the property pattern templates in both the FSA and the DNL representations and presents a detailed example for one of the patterns, using both forms. Section 4 details an example of the specification process using PROPEL. Section 5 addresses how we can incorporate scopes into the property pattern template representations. Section 6 discusses related work and Section 7 concludes with a discussion of limitations and future directions.

2. The Property Patterns

Dwyer, Avrunin, and Corbett [8-10] developed a system of property patterns to assist users of finite-state verification tools, such as SPIN [18], SMV [21], INCA [4], and FLAVERS [11]. They argued that the difficulty of writing correct properties in the various input formalisms used by such tools was a substantial obstacle to the adoption of finite-state verification technology, and they proposed the pattern system as a way to capture the experience of expert specifiers and to enable the transfer of that experience to other practitioners.

Although the input formalisms of the various finite-state verification tools, such as the temporal logics LTL and CTL [3] are very expressive, Dwyer et al. observed that nearly all the properties found in the finite-state verification literature could be classified into a small number of basic types, and suggested that a collection of patterns, which they described as "high-level, formalism-independent, specification abstractions," could assist finite-state verification practitioners in formulating most of the properties they wanted to check.

Each of the patterns describes an *intent* (the structure of the specified behavior), a *scope* (the extent of program execution over which the pattern must hold), mappings into the input formalisms for some finite-state verification tools, examples of known uses, and relationships to other patterns. For instance, the intent of the Response pattern is a cause-and-effect relationship between a pair of events or states, in which the occurrence of the "cause" or "action" leads to an occurrence of the "effect" or "response." Dwyer et al. identified five scopes, or segments of program execution, in which the specifier might want to insist that the specified intent holds. For instance, a particular Response relation might be intended to hold only while the system is executing in a certain mode, while instances of the action might require an entirely different response in other modes. The scopes are: global (the whole 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 the between scope 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 mappings to various specification formalisms involve a number of choices. For instance, in state-based formalisms, Dwyer et al. chose to take the interval in which the property is to be evaluated to be closed on the left and open on the right. Thus, the scope consists of all states beginning with the starting state and up to but not including the ending state. They chose closed-left open-right scopes because these were relatively easy to encode in specifications and were the most commonly encountered in the real properties that they had collected. They recognized, however, that other variations of the scopes might be required, such as open-left open-right scopes, and their web site [10] includes notes on how to modify the mappings to obtain such variations. These notes also discuss such issues as combinations of the patterns and which instantiations of parameters in the patterns are safe in which formalisms.

A specifier who wishes to modify a pattern, however, must have significant expertise with the particular specification formalisms utilized by the finite-state verification tool being applied. Indeed, the property patterns themselves do not highlight the choices made and the notes do not attempt to point out all plausible modifications. It is assumed that the analyst who wants to verify a particular property can identify the ways in which it might differ from the particular forms in the property pattern system and, with some assistance from the notes, make the necessary modifications. Since the target audience for the property patterns system is users of finite-state verification tools, and expertise with the specification formalisms is a prerequisite for effective use of such tools, this is not an unreasonable requirement.

In this work, however, we are concerned with eliciting precise and rigorous requirements from people who are unlikely to be fluent in temporal logics or other specification formalisms. We are thus especially interested in identifying the possible variations and determining which of these are intended. Our focus is on pointing out the various ways in which a high-level requirement might be interpreted and on helping the specifier elucidate the property by making informed choices between these interpretations.

3. PROPERTY PATTERN TEMPLATES

In our previous work with finite-state verification systems, we have found that finite-state automata, with their corresponding graphical depictions, are some of the more accessible notations for representing properties. We have also observed that many of the "shall" phrases found in requirements and specification documents seem to almost take on a template form. Thus, we wanted to see if we could marry these two notations via the property patterns. While the property pattern work included both state- and event-based formalisms, here we assume an event-based formalism.

3.1 FSA Property Pattern Templates

3.1.1 FSA Template Notation

An FSA property is defined by the tuple $<S, s, A, \Sigma, \delta>$, where S is the finite set of states, $s \in S$ is the unique start state, $A \subseteq S$ is the set of accepting states, Σ is the event alphabet, and $\delta: S \times \Sigma \rightarrow S$ is a transition function. A sequence $e_1, e_2, \ldots, e_n \in \Sigma^*$ is *accepted* by the FSA if a sequence of states $s_0, s_1, \ldots, s_n$ exists in S such that:

1. $s_0 = s$,
2. $s_n \in A$, and
3. $\delta(s_{i-1}, e_i) = s_i$ for i= 1, ..., n.

Traditionally, when depicting an FSA, states are shown as circles, the start state is denoted by an arrowhead on the circle, accepting

states are indicated by inner concentric circles, and the transitions are denoted by arrows between states indicating the direction of flow in the automaton. Each transition is labeled by one or more events from the alphabet Σ.

The FSA template notation extends the FSA property notation with the following additions:

- optional transitions,
- optionally-accepting states,
- multi-labels,
- "¬", the set complement operator, and
- ".", the wildcard character, representing all of Σ.

An optional transition is indicated by a dashed line instead of a solid line. An optionally-accepting state is denoted by a state with a dashed inner concentric circle. A multi-label is denoted by a list of alternative sets of labels, each set separated by the word "or". The "¬" operator provides a shorthand notation to indicate the complement of the given set of events with respect to the property alphabet. The "." wildcard character is a shorthand notation for expressing the set of all events in the property alphabet.

A property pattern template is *fully instantiated* when all the optional choices have been resolved and *partially instantiated* if only some of the options have been resolved. An optional transition in an FSA template will either resolve to a regular transition or it will not exist in the instantiated property. A multi-label on a regular transition will resolve to only one of its label choices in the instantiated property. A multi-label on an optional transition will resolve to at most one of its label choices; an optional transition disappears if all of the label choices in its multi-label have been eliminated from consideration. An optionally accepting state will either resolve to an accepting state or a non-accepting state. After fully instantiating an FSA template by resolving all of the options, the specifier is left with an FSA property.

3.1.2 The Response FSA Template

We illustrate how a property pattern is represented as an FSA template using the Response pattern. As noted above, the Response pattern is concerned with expressing the concept of a stimulus event that *must* be followed by a response event. An example of this property pattern as written in natural language might look something like this:

> *After the elevator button is pushed, the elevator closes its doors.*

This property looks reasonably straightforward, but a closer examination will reveal that there are many questions concerning the precise meaning that need to be answered. For example, should the doors close repeatedly if the button is pushed repeatedly? What, if anything, is allowed to occur after the button is pushed, but before the doors are closed? We find that these questions can be captured by using the extended FSA property notation with six options. These options in the Response pattern template combine to produce a total of *sixty-four* distinct variations on this property pattern. When all of the options have been decided, an FSA that represents only one of the sixty-four possible variations is created. Figure 1 shows the Response FSA template, with all of the optional components displayed.

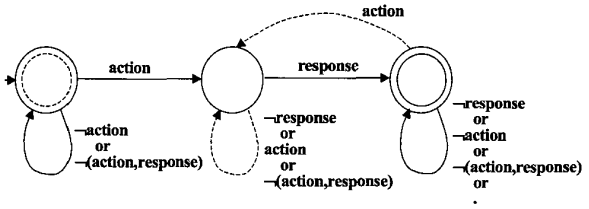

Figure 1. Response FSA Template

Let us now examine the six options in the Response pattern template in more detail. This template has two pattern parameters: **action** and **response**. These parameters are placeholders for the events in the pattern that will be specified when the property pattern template is instantiated. The six Response pattern template options are as follows:

- *Pre-arity*, which determines whether **action** may occur one time or many times before **response** does;
- *Post-arity*, which determines whether **response** may occur one time or many times after **action** does;
- *Immediacy*, which determines whether or not other intervening events may occur between **action** and **response**;
- *Precedency*, which determines whether or not **response** is allowed to occur before the first occurrence of **action**;
- *Nullity*, which determines whether or not **action** must ever occur; and
- *Repeatability*, which determines whether or not occurrences of **action** after an occurrence of **response** are required to be followed by **response**.

Each of the above options is represented by particular components in the Response FSA template, as shown in Figure 1. Pre-arity is determined by the label chosen for the self-loop on the second state. If the self-loop does not exist because all of the choices for the multi-label associated with it have been removed from consideration, or if the label on it is **¬(action,response)**, then **action** may only occur once before **response** does. Any other label choice for that multi-label will permit **action** to occur multiple times before **response** occurs. Post-arity is determined by the multi-label on the self-loop of the third state. If the label on that transition becomes **¬(action,response)** or **¬response**, then **response** may only occur once after **action** has occurred. The other two labels on that transition allow **response** to occur multiple times after **action** has occurred.

Since we assume that the alphabet may include more events than just **action** and **response**, Immediacy deals with whether or not these other events may occur at the second state. Immediacy is determined by the label chosen for the self-loop on the second state. If the self-loop does not exist because all of the choices for the multi-label associated with it have been removed from consideration, or if the label chosen for the self-loop is **action**, then other intervening events are not allowed to occur between **action** and **response**. If the label chosen for the self-loop is

¬(action,response) or **¬response**, then other intervening events are allowed to occur between **action** and **response**.

Precedency is determined by the label on the self-loop of the first state. If the label is **¬(action,response)**, then **response** may not occur before the first occurrence of **action**. Nullity is determined by whether or not the first state is an accepting state. If the inner circle exists, then the state is accepting, and **action** is not required to occur for the property to be satisfied. Repeatability is determined by the existence of the transition from the third state to the second on an occurrence of **action**. If the property is repeatable, the label on the third state's self-loop cannot include **action**.

Figure 2 shows the other FSA templates developed for PROPEL, based on the property patterns proposed by Dwyer et al. These three patterns and the Response pattern are the basic patterns that do not use any composition to express their concepts. The first pattern, Precedence, states that an **action** cannot occur until it has been preceded by the **enable** event. The second pattern, Existence, states that **action** must occur in the system execution. The last pattern, Absence, states that **action** must not occur in the system execution. Note that the Absence FSA template does not have any options.

During the process of instantiating an FSA template, the specifier must define the alphabet and associate the appropriate events with their related pattern parameters. The FSA template structure is designed to assist the specifier in asking and answering the appropriate questions and in understanding the meaning of the decisions that are made. The specifier instantiates a property pattern template until all of the options have been resolved and an FSA property representation results.

3.2 Disciplined Natural Language Templates

3.2.1 DNL Template Notation

Disciplined Natural Language (DNL) is the second representation that we propose to express the property pattern templates. As can be gathered from its name, DNL is a restricted subset of natural language that is intended to capture meanings unambiguously. This representation is not intended to stand by itself; it is meant to be used in conjunction with the FSA template representation. In expressive power, each DNL template corresponds to a single FSA template, and a fully-instantiated DNL template is mapped to a fully-instantiated FSA template. It is therefore possible to translate between the two representations and to develop them in parallel in the PROPEL process, as described in Section 4. It is hoped that a DNL property instantiated from a DNL template will improve accessibility, while the corresponding FSA property provides a rigorous and unambiguous representation.

Like FSA templates, DNL templates are designed to elucidate the decisions associated with a property pattern. Therefore, the same options that must be decided in the FSA template are options in the DNL template representation. A DNL template for a particular property pattern consists of a Core phrase and perhaps one or more subsidiary phrases. We made the decision to have multiple phrases based on our sense of what is understandable; some of the instantiated DNL properties would have resulted in long and unwieldy sentences if confined to a single phrase. The Core phrase is used to express the basic meaning of the property pattern, and may itself be parameterized, to express one or more of the options. For customization, we introduce synonymous

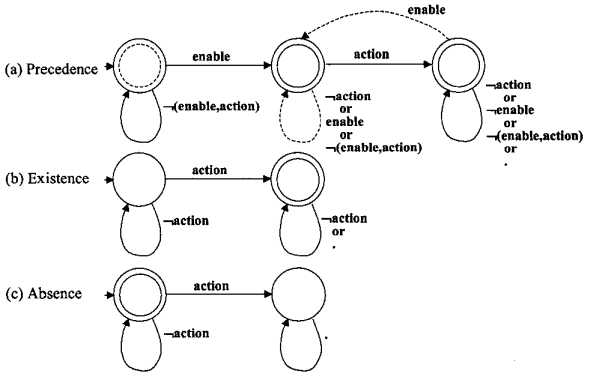

Figure 2. The Other Property Patterns' FSA Templates

choices for most of the phrases so that specifiers can select the synonym that seems most natural to the particular property that they are trying to represent.

3.2.2 The Response DNL Template

The Response DNL template consists of four phrases: the Core phrase, the Nullity phrase, the Precedency phrase, and the Repetition phrase. We provide six synonymous ways of expressing the Core phrase. In addition, the Core phrase has three options to be determined within it: Pre-Arity, Post-Arity, and Immediacy. In the cases of Pre-Arity and Post-Arity, the options have multiple synonyms associated with them. The Nullity phrase has two options, and within those there are two synonyms apiece. The Precedency phrase and the Repetition phrase have two options each and no synonyms. Figure 3 provides a full description of the four phrases and their available choices. In this figure, bold lines are used to separate options and non-bold lines are used to separate synonyms.

The Pre-Arity option provides a choice between stating that **action** can occur only once before **response** does or stating that **action** can occur multiple times before **response** does. The Immediacy option is a choice between "immediately" and "eventually," concerning whether or not **action** must be immediately followed by **response** or if other events may intervene in the sequence. The Post-Arity option is structured with exactly the same content as the Pre-Arity option, but it is placed at the opposite end of the Core phrase sentence. The Nullity phrase option provides a choice between stating that **action** must occur during the program execution or stating that **action** is not required to occur during the program execution. The Precedency phrase option is concerned with whether or not **response** is permitted to occur before the first **action** does. Finally, the Repetition phrase option determines whether or not the behavior described by the above phrases is repeatable. The DNL templates for the Precedence, Existence, and Absence property patterns are similar in structure to the Response DNL template.

A fully-instantiated DNL template results in a paragraph of natural language text that is grammatically correct, readable, and maps to one, and only one, fully-instantiated FSA property. The relationship between the options in a pattern's DNL template and those in a pattern's FSA template is not necessarily one-to-one, however, since some DNL options affect the FSA templates in more than one location. Once an option has been decided, it does

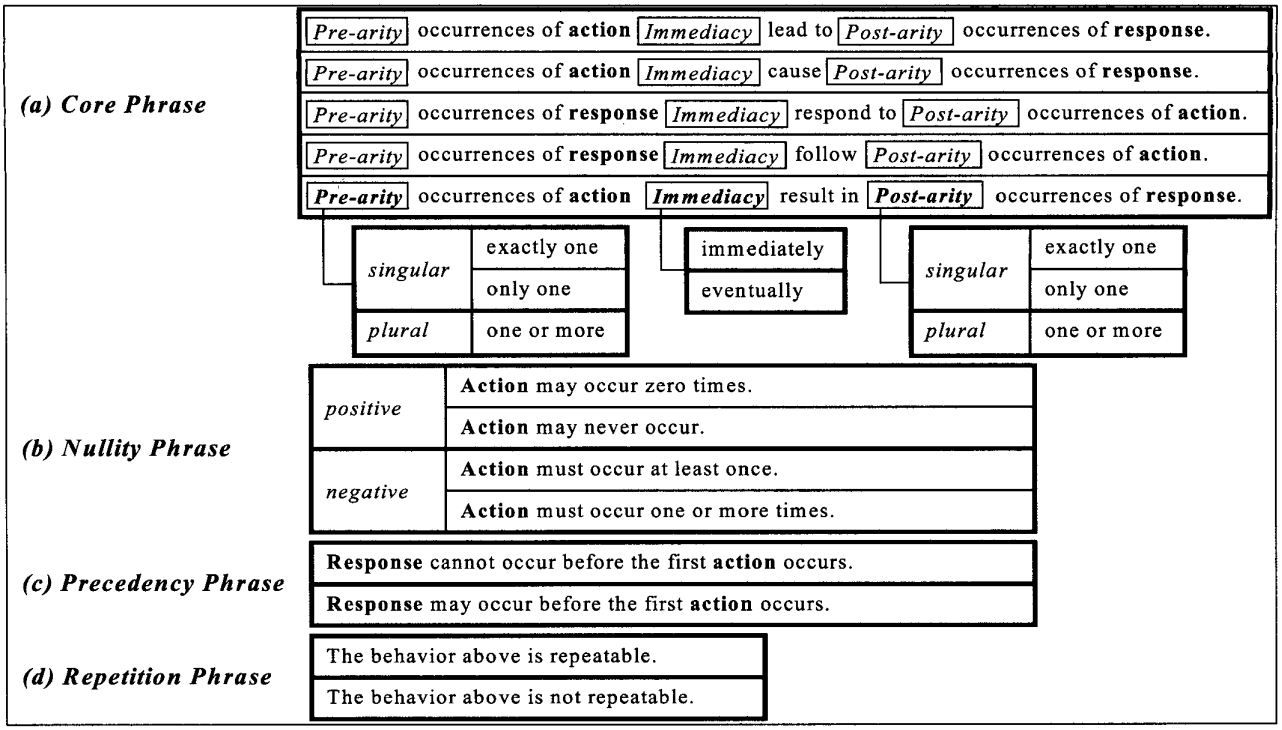

Figure 3. Response DNL Template

not matter which of the synonyms for that particular choice are selected for the final DNL property. The synonyms are available so that the specifier can customize the DNL.

Requirements documents written using PROPEL property templates can be used to produce natural language descriptions as well as FSA representations of the properties and, thus, provide rigor and precision in addition to accessibility.

4. HELPING THE SPECIFIER WRITE PROPERTIES

PROPEL provides an environment for specifying properties based on the property pattern templates. Specifiers are shown both the FSA template and the DNL template for each pattern and can select the pattern that seems most appropriate. When specifying the property alphabet or the parameters or instantiating the options, the specifier can use either the FSA or DNL template, or can change back and forth between the two representations. PROPEL keeps track of how the options in the two representations relate to each other; once an option in one representation is resolved the corresponding options in the alternative representation are also resolved. The specifier can choose to resolve the options in any order.

Suppose that the specifier has in mind the statement first shown in Section 3.1.2:

> *After the elevator button is pushed, the elevator closes its doors.*

As noted earlier, this is an example of a Response pattern, so the specifier would indicate this and then be presented with both the Response FSA template and the Response DNL template. The specifier must then determine what events the **action** and **response** pattern parameters map to in the alphabet of this property. The actual event names depend upon the event annotation mechanism for the finite-state verification system that is being used. For simplicity, let us assume that we have identified two events in the artifact that correspond to pushing a button and closing the doors, respectively. Let the alphabet of this property be *button-push*, *door-close*, and any other events of interest in the system. The specifier needs to explicitly state which of the events corresponds with which of the pattern parameters. Figure 4a shows the initial Response FSA template with the pattern parameters replaced by their respective events.

The next step for the specifier might be to select how to express the basic stimulus-response concept in DNL. For instance, the specifier could select the Core phrase synonym "action results in response" (see Figure 3 for the full list of Core phrase possibilities). The following sentence would then be displayed as the Core phrase in the partially-instantiated DNL property:

Pre-Arity occurrences of *button-push* Immediacy result in Post-Arity occurrences of *door-close.*

The options in the boxes in the Core phrase need to be determined, and at this point the specifier could change the setting of the options by either manipulating the associated FSA template or by choosing between the DNL options available. Let us assume that the specifier decides to use the FSA template to determine the Pre-Arity and the Post-Arity of the property. Figure 4b shows the partially-completed FSA template where Pre-Arity has been determined such that *button-push* may occur one or more times before *door-close* is required to occur, and Post-Arity has been determined such that after *button-push* has occurred, *door-close* may occur only once. For the Pre-Arity option, the FSA template has been changed such that the multi-label on the second state's self-loop no longer contains the ¬(*button-push, door-close*) possibility. For the Post-Arity option, the FSA template cannot allow *door-close* to occur on the third state's self-loop, and thus the ¬*button-push* and "." possibilities are removed from the multi-label on that transition. At this point, the DNL Core phrase looks like this:

> One or more occurrences of *button-push* Immediacy result in only one occurrence of *door-close.*

In Figure 4c, the specifier has decided to make the property repeatable and indicated that setting in the FSA template by making the transition from the third state to the second state on *button-push* a solid line. This change also has an effect on the multi-label on the self-loop of the third state, since *button-push* cannot be allowed to occur on that transition. Therefore, the multi-label on that self-loop is resolved to the label ¬(*button-push, door-close*). PROPEL reflects this change in the DNL representation by resolving the Repetition phrase to "The behavior above is repeatable."

To finish the Core phrase in the DNL, the specifier could decide to determine the *Immediacy* option by choosing "eventually" for the DNL option there. This affects the FSA template in two ways. First, intervening events are now allowed to occur between *button-push* and *door-close*, so the self-loop on the second state becomes a solid-line transition. Second, the multi-label on that self-loop is resolved to the one label, ¬*door-close*, because *button-push* is now allowed to occur on this transition, since the Pre-Arity was set to be "one or more times" in a previous step, and because using the set complement operator allows intervening events other than *button-push* to occur on this transition. Two phrases in the Response DNL template have now been completed: the Core phrase and the Repetition phrase. The DNL template now looks like this:

> One or more occurrences of *button-push* eventually result in only one occurrence of *door-close.*
> Nullity phrase
> Precedency phrase
> The above behavior is repeatable.

Assume the next option selected is the question of Nullity; that is whether or not *button-push* must occur in the program execution. The specifier decides to set this option by making the optionally-accepting state in the FSA template an accepting state. PROPEL reflects this change in the DNL representation by resolving the

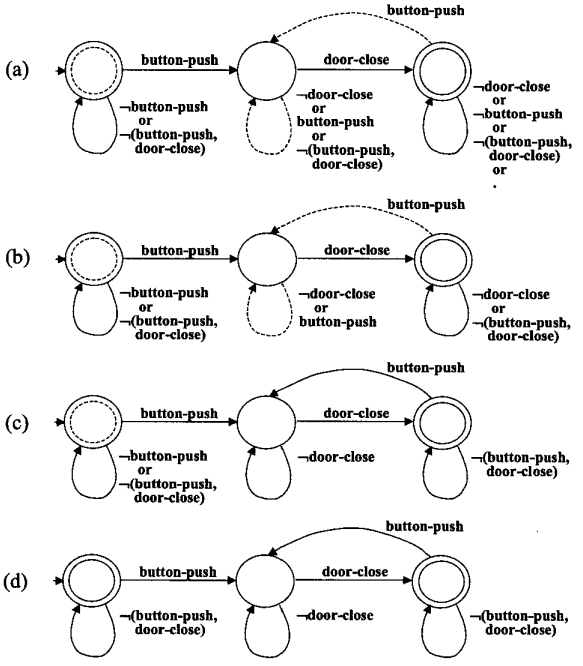

Figure 4. An Example of the PROPEL Process

Nullity phrase to "*button-push* may occur zero times." The DNL template provides a synonym to this choice, and the specifier can later decide whether or not to change which synonym is used.

The last option set is Precedence, and the specifier decides to use the FSA template to make the choice. The multi-label on the first state's self-loop is resolved to be ¬(*button-push,door-close*), which does not allow *door-close* to occur until *button-push* has occurred at least once. The accompanying DNL option results in the statement, "*door-close* cannot occur before the first *button-push* occurs," being added to the DNL template.

After fully-instantiating all the options, the FSA template is resolved to an FSA property and the DNL property is resolved to a completed natural language paragraph. The final FSA property for the elevator example is shown in Figure 4d and the final DNL property could be:

> One or more occurrences of *button-push* eventually result in only one occurrence of *door-close. button-push* may occur zero times. *door-close* cannot occur before the first *button-push* occurs. The behavior above is repeatable.

Any option in a property in PROPEL can be unset and reselected if the option needs to be changed, and the DNL can be customized by choosing a different synonym at any time. Thus, the process is designed to help specifiers ask questions about their assumptions and elucidate the meaning of a property. The specifier could go through this process in a different order than has been described above and could make different decisions about when to use the FSA and DNL representations.

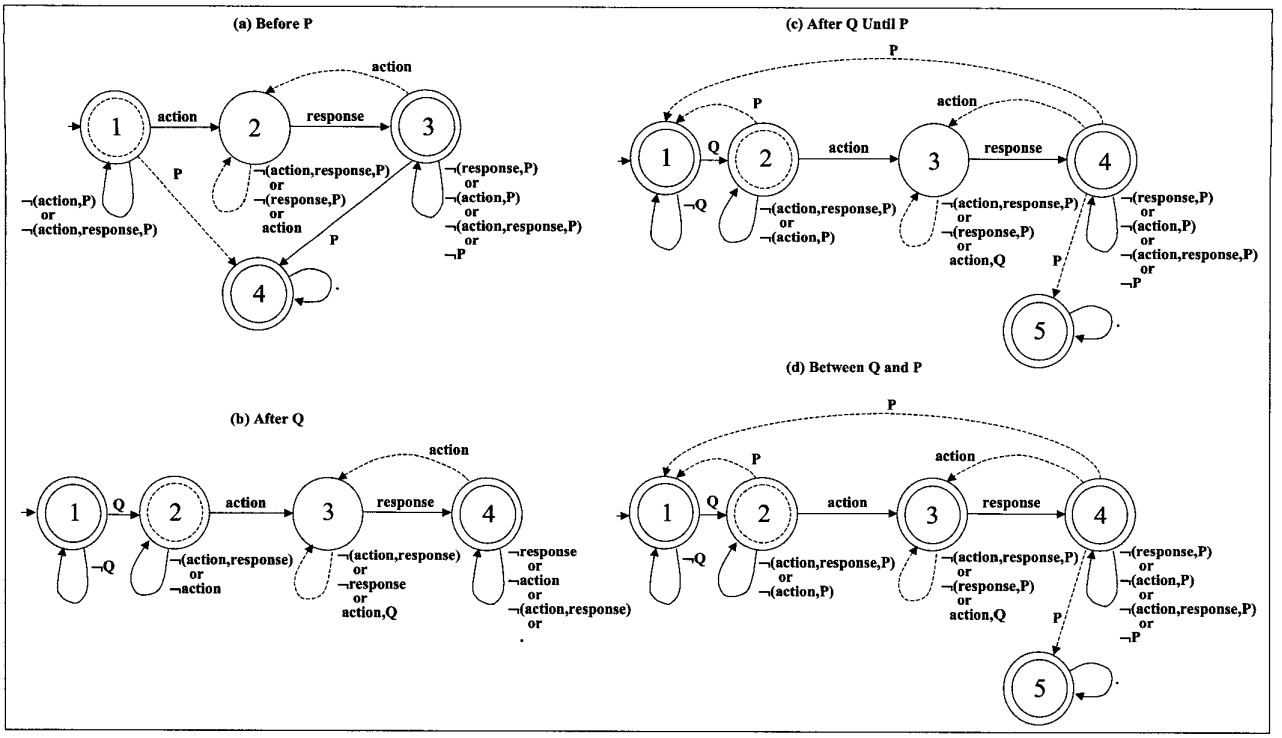

Figure 5. Four Scopes Applied to the Response FSA Template

5. INCORPORATING SCOPES

So far, we have discussed how to develop the intent (the structure of the behavior specified) of a property. In this section, we discuss the definition of a scope and how it is applied to the property pattern template. As mentioned above, the patterns of Dwyer et al. have scopes that describe the extent of system execution over which the pattern must hold. For example, a specifier might want to say:

> *Between pushing the button and arriving at the requested floor, the elevator does not change direction.*

A scope can be used to specify when it is important that the property holds.

Dwyer et al. identified five basic scopes and defined whether the scope was closed or open on either end for each pattern. Most event-based formalisms use some version of an interleaved model of concurrent computation. In such formalisms, two events cannot coincide. In the property patterns system, therefore, event-delimited scopes are open at both ends; an event that occurs within the scope cannot occur at the same time as an event that marks the beginning or end of the scope. We make the additional restrictions that the set of delimiter events defining the scope is disjoint from the alphabet of the FSA defining the intent and that, if the scope has both starting and ending delimiters, these are distinct. These restrictions seem reasonable and greatly simplify the application of a scope to an intent.

With these restrictions, a scope may be characterized by three attributes: the starting delimiter, the ending delimiter, and whether or not the scope may occur repeatedly within the system execution. We represent the starting delimiter as the pattern parameter **Q**, and the ending delimiter as the pattern parameter **P**. Thus, the five basic scopes become:

- *Global*, which is over the entire system execution. This scope does not have delimiters, and it is not repeatable.
- *Before P*, which is concerned with the event sequence up to the first occurrence of **P**. This scope only has a ending delimiter, and it is not repeatable.
- *After Q*, which is concerned with the event sequence after the first occurrence of **Q**. This scope only has an starting delimiter, and it is not repeatable.
- *Between Q and P*, which is concerned with the event sequence after an occurrence of the starting delimiter, **Q**, and before an occurrence of the ending delimiter, **P**. **P** is required to occur for this scope to exist. This scope may occur repeatedly over the course of the system execution.
- *After Q Until P*, which is concerned with the event sequence after an occurrence of the starting delimiter, **Q**, and before an occurrence of the ending delimiter, **P**. **P** is not required to occur for this scope to exist. This scope may occur repeatedly over the course of the system execution.

With the restrictions noted above, there is limited interaction between the intent and the scope. Thus, these restrictions allow us

to propose a rather straightforward way to apply scopes to property pattern templates. We now describe how the templates can be extended to take scopes into account.

5.1 Applying Scopes to the FSA Templates

A scope can be applied to an intent by adding additional states that we call "scope states" and by adding transitions between the scope states and the states of the FSA that represent the intent. These transitions are labeled by the scope delimiters, **Q** and **P**. We add these scope delimiters to the alphabet, Σ. A scope state that has a self loop for every event in the alphabet is called a *trap state*. Figure 5 shows the Before P, After P, After Q Until P, and Between Q and P scopes added to the Response FSA template. The Global scope is not shown since it makes no visible changes to the FSA template. We have added numbers to the states in Figure 5 for easy reference. For the sake of brevity, instead of separately showing the self-loops on the states in the intent that are added for occurrences of the delimiters, we have changed the multi-labels on the existing self-loops to reflect these additions where possible. We have also not shown the transitions that go to a non-accepting trap state; when a transition is not provided that explicitly allows a delimiter to occur, it should be assumed that an occurrence of that delimiter puts the FSA into a non-accepting trap state. In the remainder of this section, we explain the additions needed to apply each scope to the Response pattern. Scopes are added to the intents of the other property patterns in much the same way.

5.1.1 The Before P Scope

We interpret the Before P scope to mean that the scope begins at the start of the program execution, so there is no starting delimiter, and that the ending delimiter is the first occurrence of **P** in the program execution. Subsequent occurrences of **P** are ignored since this scope is not repeatable. When we apply the Before P scope to the intents, we must determine at each state what the effect will be of encountering the ending delimiter at that point in the sequence. Recall that the intent of the Response property is that an occurrence of **action** must be followed by an occurrence of **response**.

In the first state, labeled "1" in Figure 5a, an occurrence of **P** results in a scope that has been ended before the intent of the property has been entered. This is called an *empty scope*. An empty scope is handled differently for each of the property patterns. For instance, the Absence property holds if the scope is empty, since that is actually the meaning of the Absence property pattern. Whether or not a Response property holds if the scope is empty, however, depends on the setting of the Nullity option, which determines whether or not **action** must occur at all. If **action** is not required to occur, then the first state, which is also the start state, will be an accepting state. If **P** occurs at this point and **action** is not required to occur, then this sequence is not a violation of the property and the empty scope does not prevent the property from holding. The optional transition that goes to the accepting trap state, labeled "4", would become a regular transition. If **action** is required to occur, then that first state will not be an accepting state. In this case, the occurrence of **P** when the FSA is in the first state puts the FSA into a non-accepting trap state because the scope would be ended before the intent's first requirement was met; this sequence would be a violation of the property. The optional transition that goes to the accepting trap state would not exist. As noted above, we do not show the transitions that go to a non-accepting trap state.

When the FSA is in the second state, an **action** has occurred that is not yet followed by a **response**. At this point, **response** must occur before a **P** ends the scope, or the property is violated. An occurrence of **P** when the FSA is in the second state therefore puts the FSA into a non-accepting trap state.

The third state is an accepting state and an occurrence of **P** at that point in the sequence could not violate the property. From this state, **action** is not required to occur and there is no occurrence of **action** that is waiting for an occurrence of **response**. If the scope is ended when the FSA is in the third state, the property holds. Since the Before P scope is not repeatable, we add a transition on an occurrence of **P** that goes from the third state to an accepting trap state.

Applying a scope to the Response FSA template also affects the multi-labels. Because the set complement operator means that everything that is not in the set specified by the label choice is accepted, **P** needs to be added to each of those labels in which the set complement operator is used. Note that the final label choice in the multi-label on the third state's self-loop is changed from the whole set of events (".") to the set of events excluding **P** ("¬**P**").

5.1.2 The After Q Scope

We interpret the After Q scope to mean that the scope is not started until the first occurrence of the starting delimiter, **Q**, in the system execution. Subsequent occurrences of **Q** are ignored since this scope is not repeatable. The scope is not ended until the execution ends, so there is no ending delimiter. As with the Before P scope, when we apply the After Q scope to the intents, we must determine at each state what the effect will be of encountering the ending delimiter at that point in the sequence.

The first state in Figure 5b is a scope state that is added to the Response FSA template. An occurrence of **Q** at this state would begin the scope and after this point the intent of the property would be required to hold. As is shown with the self-loop that is labeled "¬**Q**" on state 1, all events that occur before the starting delimiter are ignored. An occurrence of **Q** at any of the other states would have no effect on the property, because after the occurrence of the starting delimiter all subsequent occurrences of **Q** are ignored. Therefore, all of the transitions on an occurrence of **Q** from states 2, 3, and 4 are self-loops on those states. As was stated in Section 5.1, instead of separately showing the self-loops on the states in the intent that are added for occurrences of the delimiters, we have changed the multi-labels on the existing self-loops to reflect these additions where possible. Because the set complement operator means that everything that is not in the set specified by the label choice is accepted, the only place that **Q** must be explicitly added is the multi-label on the self-loop on state 3. It is added to the label choice that only accepted occurrences of **action**, and that label choice becomes "**action, Q**".

5.1.3 The After Q Until P Scope

We interpret the After Q Until P scope to mean that the starting delimiter is an occurrence of **Q** and the ending delimiter is an occurrence of **P**. This scope can be repeated; whether or not it is repeatable is an option for the specifier to determine. For now, consistent with Dwyer et al., we assume that a scope with multiple occurrences of **Q** is ended by a single occurrence of **P**. An

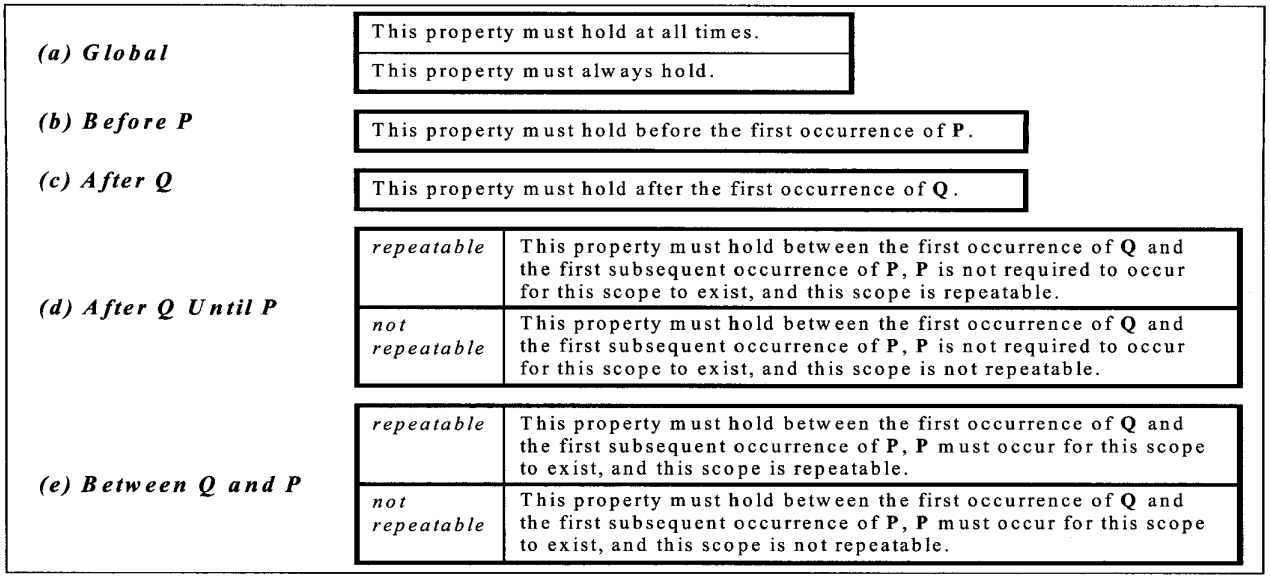

Figure 6. Scope Phrase DNL Options

occurrence of **P** that is not preceded by an unended scope is ignored. Given this interpretation, when we apply the After Q Until P scope to the intents we must determine at each state what the effect will be of encountering the ending delimiter at that point in the sequence.

The first state in Figure 5c is a scope state that is added to the Response FSA template. An occurrence of **Q** at this state would start the scope and after this point the intent of the property would be required to hold. As is shown with the self-loop that is labeled "¬**Q**" on state 1, all events that occur before the starting delimiter are ignored. An occurrence of **P** at state 1 would also be ignored, since an occurrence of **Q** has not yet started the scope.

In the second state, an occurrence of **Q** is ignored, because the scope has not yet been ended by an occurrence of **P**. An occurrence of **Q** is on a self-loop on this state and it is therefore incorporated into the multi-label on the self-loop that already exists on this state. **Q** does not need to be explicitly shown in either of the label choices on this multi-label because they are both expressed with a set complement operator.

An occurrence of **P** in state 2 again brings up the issue of what to do with an empty scope, as was discussed in section 5.1.1. If state 2 is accepting, then the optional transition from state 2 to state 1 will be a regular transition. If state 2 is not accepting, then the transition on an occurrence of **P** from state 2 will go to a non-accepting trap state.

In the third state, an occurrence of **Q** is ignored, because the scope has not yet been ended by an occurrence of **P**. An occurrence of **Q** is on a self-loop on this state and it is therefore incorporated into the multi-label on the self-loop that already exists on this state. The only place in which **Q** must be explicitly added to the multi-label on the self-loop on state 3 is the label choice that only accepted occurrences of **action**. That label choice becomes "**action, Q**".

When the FSA is in state 3, an **action** has occurred that is not yet followed by a matching **response**. At this point, **response** must occur before a **P** ends the scope, or the property is violated. An occurrence of **P** when the FSA is in state 3 therefore puts the FSA into a non-accepting trap state.

In the fourth state, an occurrence of **Q** is ignored, because the scope has not yet been ended by an occurrence of **P**. An occurrence of **Q** is on a self-loop on this state and it is therefore incorporated into the multi-label on the self-loop that already exists on this state. **Q** does not need to be explicitly shown in any of the label choices on this multi-label because they are all expressed with a set complement operator.

Figure 5c shows two optional transitions on an occurrence of **P** in state 4. If the specifier makes this scope repeatable, then an occurrence of **P** in state 4 would end the scope and allow the scope to be restarted by a subsequent occurrence of **Q**. The optional transition between state 4 and state 1 would then become a regular transition and the optional transition between state 4 and state 5 would not exist. If the specifier makes this scope not repeatable, then an occurrence of **P** in state 4 would put the FSA into the accepting trap state, labeled "5." In this case, the transition between state 4 and state 5 would become a regular transition and the optional transition between state 4 and state 1 would not exist.

5.1.4 *The Between Q and P Scope*

The Between Q and P Scope is identical to the After Q Until P scope except for one important difference. Figure 5d shows the difference: state 3 is accepting. The reason for this change is that the definition of the Between Q and P scope requires that both delimiters occur, whereas the After Q Until P scope does not. The Between Q and P scope does not exist unless both of its delimiters occur. What this means is that if **P** does not occur, the intent of the property could be violated and yet the property as a whole would not be violated because it is not within an existing scope

when the violation happens. This is a "look-back" scope, where once a **P** occurs, the sequence of events up to that point must conform to the sequence required by the property, and only then can a violation of the property be determined. Applying this scope also affects the internal structure of the FSA templates for each of the other patterns.

5.2 Applying Scopes to the DNL Templates

For DNL templates, applying a scope to the intent of a property is relatively straightforward: we add another phrase to the template: the Scope phrase. Therefore, the complete Response DNL template is:

Core phrase
Nullity phrase
Precedency phrase
Repeatability phrase
Scope phrase

The Scope phrase can be resolved to one of the five scopes. Figure 6 shows the choices available for each scope in the DNL. The only option to be addressed for scopes is whether or not they are repeatable. Scope repeatability determines whether or not an occurrence of **Q** after the first occurrence of **P** is required to be followed by another occurrence of **P** for the scope to be closed. The Global, Before P, and After Q scopes cannot be made repeatable. This option is expressed in the DNL template by the choices provided for the After Q Until P scope and the Between Q and P scope.

5.3 Using Scopes in PROPEL

The example in Section 4 showed how PROPEL would be used without taking scopes into account. Actually, the specifier must select a scope as well an intent and must then resolve any options associated with their combination.

6. RELATED WORK

The PROPEL approach described in this paper builds directly on the property patterns [8]. That work identified commonly-occurring types of specifications and attempted to provide users of finite-state verification tools with high-level, formalism-independent abstractions for dealing with those types. This work has been extended in a number of directions. For instance, these patterns form the basis of the extensible specification language in the Bandera system [5, 6], and Paun and Chechik [23] have extended the patterns to deal with events in a state-based formalism. A number of other researchers have used templates or patterns in the construction of both requirements and properties for finite-state verification. For instance, van Lamsweerde and his co-authors [7, 20] have suggested using a library of refinements to construct detailed requirements from goals. The correctness of these refinements is verified in a formal logic. The Attempto Controlled English project [13, 14] offers annotated templates to guide non-expert users, and the Cico/Circe [1] tool includes suggested phrases for expressing relationships between artifacts. The FormalCheck [12] finite-state verification tool uses templates to formulate the properties to be checked.

Other techniques, such as various tabular notations, have been aimed at providing requirements that are both accessible and suitable for formal analysis. The work of Heninger and her co-authors on the A-7E project [17] focused on expressing properties with condition- and event-tables. Heitmeyer and her co-authors (e.g., [16]), have built a variety of tools for checking consistency, completeness, and safety properties of requirements expressed in the tabular SCR notation. The Requirements State Machine Language [19], which provides a tabular notation for the guarding conditions of transitions to help make the requirements accessible to domain experts, supports similar analyses [15]. These approaches are general formalisms for expressing requirements, while the PROPEL approach aims at elucidating common properties that arise in finite-state verification.

Some research, such as the Attempto Controlled English project, Cico/Circe, NLIPT [22], and the work of Bryant [2], attempts to construct formal specifications from natural language requirements. The use of natural language in the work described here is much less ambitious. PROPEL provides both disciplined natural language and FSA representations, and allows the specifier to move back and forth between them in order to help make the formal specifications more understandable and accessible, but this work does not attempt to understand natural language, even in restricted domains.

7. CONCLUSIONS

With PROPEL, users are provided with templates for the most common property patterns described in Dwyer et al. These templates are presented in an extended finite-state automaton notation and as natural language phrases, both of which explicitly indicate the options that must be considered. We hypothesize that this two-pronged approach will help specifiers elucidate the precise meaning of the properties that they are expressing. We are currently implementing the PROPEL system so that specifiers are presented with both notations and can move between them while instantiating the property templates incrementally. We believe that this approach is an effective way to achieve both accessibility and rigor in property specifications.

There are a number of interesting directions that we intend to explore in future work. We want to study compositions of specification patterns, including arbitrary compositions and restricted compositions such as chaining. We intend to investigate new ways of composing scopes, such as nested scopes, and the possibility of loosening the restrictions on their alphabets. We also plan to extend the templates to include state-based and mixed event- and state-based notations. Although we have shown specific natural language phrases, one could argue that the resulting properties could be stated better. It would be reasonable to have natural language experts help define the phrases associated with each property. Another possible direction to explore is developing these properties in a non-interleaved model of concurrent computation. Most importantly, we plan to evaluate this approach by applying it to some industrial applications. Although we have applied this approach to several properties and been pleased with the results, we need to undertake a careful evaluation.

8. ACKNOWLEDGEMENTS

We thank Jamieson M. Cobleigh for his helpful comments.

This research was partially supported by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231, by the National

Science Foundation under Grant CCR-9708184, by the U.S. Army Research Laboratory and the U.S. Army Research Office under Agreement DAAD190110564, and by IBM Faculty Partnership Awards. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U. S. Army, the U.S. Dept. of Defense, the U.S. Government, the National Science Foundation, or of IBM.

9. REFERENCES

[1] Ambriola, V. and Gervasi, V. "Processing Natural Language Requirements," in 12th Int. Conference on Automated Software Engineering. 1997. Lake Tahoe, NV. p.36-45.

[2] Bryant, B. "Object-Oriented Natural Language Requirements Specification," in ACSC 2000, the 23rd Australasian Computer Science Conference. 2000. Canberra, Australia.

[3] Clarke, E.M., Grumberg, O., and Peled, D.A., "Model Checking." 2000: MIT Press.

[4] Corbett, J.C. and Avrunin, G.S., "Using Integer Programming to Verify General Safety and Liveness Properties." Formal Methods in System Design, 1995. 6: p. 97-123.

[5] Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach', S., Pasareanu, C.S., Zheng, R., and Zheng, H. "Bandera: Extracting finite-state models from Java source code," in 22nd Int. Conference on Software Engineering. 2000. Limerick, Ireland. p.439-448.

[6] Corbett, J.C., Dwyer, M.B., Hatcliff, J., and Robby. "A Language Framework for Expressing Checkable Properties of Dynamic Software," in SPIN Software Model Checking Workshop. 2000. Stanford, CA. p.205-223.

[7] Darimont, R. and van Lamsweerde, A. "Formal Refinement Patterns for Goal-Drive Requirements Elaboration," in 4th ACM SIGSOFT Symp. on the Foundations of Software Engineering. 1996. San Francisco, CA. p.179-190.

[8] Dwyer, M.B., Avrunin, G.S., and Corbett, J.C. "Patterns in Property Specifications for Finite-State Verification," in 21st Int. Conference on Software Engineering. 1999. Los Angeles, CA. p.411-420.

[9] Dwyer, M.B., Avrunin, G.S., and Corbett, J.C. "Property Specification Patterns for Finite-state Verification," in 2nd Workshop on Formal Methods in Software Practice. 1998. Clearwater Beach, Florida. p.7-15.

[10] Dwyer, M.B., Avrunin, G.S., and Corbett, J.C. "Specification Patterns Web Site." http://www.cis.ksu.edu/santos/spec-patterns/.

[11] Dwyer, M.B. and Clarke, L.A. "Data Flow Analysis for Verifying Properties of Concurrent Programs," in 2nd ACM SIGSOFT Symp. on the Foundations of Software Engineering. 1994. New Orleans, LA. p.62-75.

[12] FormalCheck. "Web Site." http://www.cadence.com/datasheets/formalcheck.html

[13] Fuchs, N.E., Schwertel, U., and Schwitter, R. "Attempto Controlled English -- Not Just Another Logic Specification Language," in 8th Int. Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'98). 1998. p.1-20.

[14] Fuchs, N.E. and Schwitter, R. "Attempto Controlled English (ACE)," in CLAW 96, the 1st Int. Workshop on Controlled Language Applications. 1996.

[15] Heimdahl, M.P.E. and Leveson, N.G., "Completeness and Consistency in Hierarchical State-Based Requirements." IEEE Transactions on Software Engineering, 1996. 22(6): p. 363-377.

[16] Heitmeyer, C.L., Jeffords, R.D., and Labaw, B.G., "Automated Consistency Checking of Requirements Specifications." ACM Transactions on Software Engineering and Methodology, 1996. 5(3): p. 231-261.

[17] Heninger, K., Parnas, D.L., Shore, J., and Kallander, J., "Software Requirements for the A-7E Aircraft." 1978, Tech. Rep. 3876. Naval Research Laboratory: Washington, D.C.

[18] Holzmann, G.J., "The Model Checker SPIN." IEEE Transactions on Software Engineering, 1997. 23(5): p. 279-294.

[19] Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., and Reese, J.D., "Requirements Specification for Process-Control Systems." IEEE Transactions on Software Engineering, 1994. 20(9): p. 684-707.

[20] Massonet, P. and van Lamsweerde, A. "Analogical Reuse of Requirements Frameworks," in RE '97 – 3rd Int. Conference on Requirements Engineering. 1997.

[21] McMillan, K.L., "Symbolic Model Checking: An Approach to the State Explosion Problem." 1993, Boston, MA: Kluwer Academic Publishers.

[22] Michael, J.B., Ong, V.L., and Rowe, N.C. "Natural-Language Processing Support for Developing Policy-Governed Software Systems," in 39th Int. Conference on Technology for Object-Oriented Languages and Systems. 2001. Santa Barbara, CA, USA.

[23] Paun, D.O. and Chechik, M. "Events in Linear-Time Properties," in 4th Int. Conference on Requirements Engineering. 1999. Limerick, Ireland.