From Natural Language Requirements to Rigorous Property Specifications

Rachel L. Smith, George S. Avrunin, Lori A. Clarke
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003, USA
{rasmith, avrunin, clarke}@cs.umass.edu

***Abstract* – Property specifications concisely describe selected aspects of what a software system is supposed to do. It is surprisingly difficult to write these properties correctly. Although there are rigorous mathematical formalisms for representing properties, 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. The PROPEL tool 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 "disciplined" natural language, decision trees, and finite-state automata, allowing the developer to easily move between these representations.**

I. 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. 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. 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 formalisms and instead write requirements in natural language, UML state diagrams, and other less precise notations. These representations seem to be more accessible to practitioners, but they are often verbose or contain imprecise—and sometimes ambiguous and inconsistent—descriptions of the system. Thus they are of limited value when doing consistency checking, design and implementation analysis, or testing and verification of the system. In this situation, the requirements often do not provide enough value to warrant the investment put into creating and maintaining them.

What is needed is a property specification approach that bridges the gap between informal intent and precise specification. Our approach aims to do this by encouraging developers to think about the issues involved in specifying their properties precisely and by providing representations that are easy to use and understand. 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 property patterns can be instantiated with specific events or states and then mapped to several different formalisms. Our approach focuses on property templates that extend the basic property patterns with alternative options that are explicitly shown to the developer. These options are designed to assist the developer in understanding the questions about their property that need to be considered.

We provide three different notations to represent properties and the questions that can be asked about them: disciplined natural language (DNL) templates, an extended finite-state automaton (FSA) representation, and a decision tree (DT) representation. The DNL and DT representations should appeal to those developers who prefer a natural language descriptions. 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 analyses. We believe that providing developers with the ability to view all these representations simultaneously and select the available options from any representation will help them to elucidate the desired property. We are currently developing a system, called PROPEL, for "PROPerty ELucidation," that provides support for specifying properties based on the property templates, using these complementary representations.

This paper describes property templates and the capabilities of the PROPEL tool. The next section of the paper briefly explains the concerns that motivated the extension made to the original property patterns to express them as templates. Section 3 describes the property templates in the FSA, DNL, and DT representations. Section 4 presents a detailed example of using the PROPEL tool to specify one of the patterns. Section 5 discusses related work and Section 6 concludes with a discussion of limitations and future directions.

II. 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 [3], and FLAVERS [7]. They proposed the pattern system, modeled on Design Patterns [14], as a way to leverage the experience of system developers by capturing a description of good solutions to

recurring design problems. 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 parameterizable 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 a *behavior* (the structure of the specified constraint), a *scope* (the extent of program execution over which the behavior 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 behavior 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." A particular Response relation might be intended to hold only while the system is executing in a certain mode, or scope, while instances of the action might require an entirely different response in other scopes. 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 pattern system gives mappings from behavior-scope combinations to several formal notations (e.g., regular expressions, various temporal logics, etc.) The mappings involve a number of choices and the property pattern web site [10] includes notes on how to modify the mappings to obtain useful 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. 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 a developer who wishes to modify a pattern has significant expertise with the particular specification formalisms utilized by the finite-state verification tool being applied.

As an example of the subtle, but important, possible variations on a basic property pattern, consider the following property, as expressed in natural language:

> *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? Can the doors close without the button being pushed? Does the button have to be pushed at all?

In this work, 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 enabling the developer to determine which of these are intended. Our focus is on helping the developer elucidate the property by making informed choices between these interpretations.

III. PROPERTY TEMPLATES

In our previous work with finite-state verification systems, we have found that finite-state automata, with their corresponding graphical depictions, are somewhat more accessible than other mathematical 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 and extend each of the basic property patterns.

A. Finite State Automata Templates

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

- optional transitions,
- optionally-accepting states,
- multi-labels,
- "¬", the set complement operator, and
- ".", the wildcard character, representing the property's entire alphabet.

We will illustrate these notational additions in the example described in Section 4.

A property template is *fully instantiated* when all the optional choices have been resolved and *partially instantiated* if some unresolved options remain. The property templates rely on pattern parameters; during the process of instantiating an FSA template, the developer must define the alphabet and associate the appropriate events with their related pattern parameters. The FSA template structure is designed to assist the developer in asking and answering the appropriate questions and in understanding the meaning of the decisions that are made. After fully instantiating an FSA template by resolving all of the options, the developer is left with an FSA representation of their property.

B. Disciplined Natural Language Templates

The DNL is a restricted subset of natural language. This representation is not intended to stand by itself; it is meant to be used in conjunction with the FSA template representation. It is hoped that a DNL property instantiated from a DNL template will improve accessibility, while the corresponding FSA property provides a precise semantic interpretation.

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. The DNL template for a particular property pattern consists of a Core phrase and perhaps one or more subsidiary phrases. The Core phrase is used to express the basic meaning of the property pattern and may be parameterized to express one or more of the options. For customization, we introduce synonymous choices for most of the phrases so that developers can select the synonym that seems most natural to the particular property that they are trying to represent.

It is therefore possible to translate between the two representations and to develop them in parallel using PROPEL, as described in Section 4.

C. Decision Tree Templates

The FSA and DNL templates, as described above, assume that the developer has chosen a particular property pattern, after which the selected property representation can help guide the decision-making process from that point. The DT representation is somewhat more flexible and can be used to assist the developer in deciding which property pattern is desired.

In PROPEL, there are four basic event-based behaviors, which are partially represented in the initial DT template given below:

How many events are in your behavior?
- One Event:
  - The event must happen.
  - The event must NOT happen.
- Two Events:
  - The first event causes the second event to happen.
  - The first event enables the second event to happen.

Using the DT template given above, the developer chooses only one of the four behaviors, and subsequently must answer questions to determine more precisely the property that is desired. As an example of a subsequent DT template, the developer may choose the third behavioral pattern (Response) and substitute the pattern parameters **action** and **response** for the first event and the second event, respectively. The resulting Response DT template is given below. Similar DT templates have been developed for the other behavioral patterns and the scopes.

Action causes **response** to happen.
- Requiring **action** to occur:
  - **Action** must occur at least once
  - **Action** might never occur
- Allowing **response** to occur before **action**:
  - **Response** may occur before **action**
  - **Response** must not occur before **action**
- Allowing intervening events:
  - No other events may occur between **action** and **response**
  - Other events may occur between **action** and **response**
- How many times may **action** occur before **response** does?
  - **Action** may only occur once before **response** does
  - **Action** may occur one or more times before **response** does
- How many times may **response** occur after **action** does?
  - **Response** may only occur once after **action** does
  - **Response** may occur one or more times after **action** does
- Repeating the behavior:
  - The behavior described above may repeat.
  - The behavior described above may not repeat.

IV. SPECIFYING PROPERTIES

Suppose that the developer has in mind the statement first shown in Section 2:

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

The developer creates a new property in the tool by choosing one of the four basic event-based behaviors that were presented in the initial DT template. The first question that the developer must answer in that DT template is whether the property is concerned with one or two events. After the developer has made that choice, further sub-questions in the form of sentences appear. Figure 1 shows a screen capture of the tool's GUI for this part of the process. In the figure, the developer decides that the elevator property has two events and that the first event causes the second event to occur.

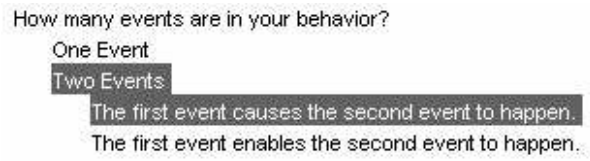

Figure 1: Choosing the Response Property Template

After the developer has selected which of the property templates best describes the type of behavior the new property should express, in this case the Response property template, the tool presents both the FSA template and DNL template for further editing. The developer can then determine what events the Response property template's **action** and **response** parameters map to in the alphabet of this property. For simplicity, let us assume that pushing a button and closing the doors are the only events of interest and that they correspond to the events *button-push* and *door-close*, which are substituted for the parameters **action** and **response**, respectively.

Figure 2 shows a screen capture of the PROPEL tool with two main windows open: the Property Views and the Alphabet Views. The Property Views window displays the initial Response FSA template (shown in the figure as the "Graphical View") and the initial Response DNL template (shown in the figure as the "Disciplined English View"), with the pattern parameters replaced by their respective specified events. The Alphabet Views window provides a place for the developer to edit the pattern parameters using the Formal Parameter View and the property's alphabet using the Alphabet Manager.

At this point, the developer is shown both the FSA template and the DNL template for the selected pattern. When

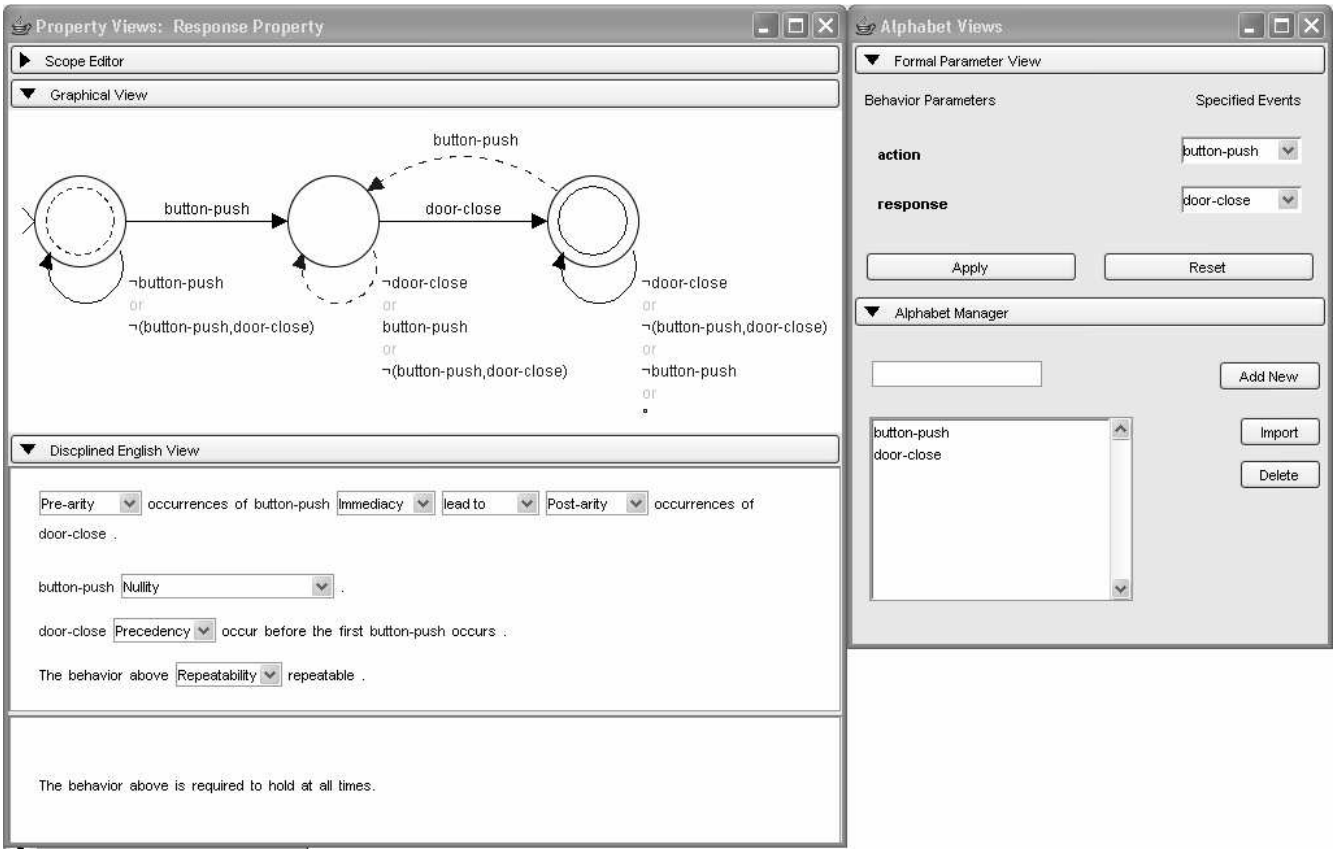

Figure 2: The Initial Response FSA and DNL Templates with Specified Events

instantiating the options, the developer can use either template, or change back and forth between the two representations. The PROPEL tool 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 developer can choose to resolve the options in any order.

Let us assume that the developer decides to use the DNL template to begin editing the property. The drop-down menus in Figure 3 indicate the options in the Response property template that need to be decided to fully instantiate a Response property. The Pre-arity option is concerned with the question of how many occurrences of *button-push* are allowed to occur before the first occurrence of *door-close*. As is shown in Figure 3, the developer can answer this question by opening the drop-down menu labeled "Pre-arity" and selecting the desired choice. It should be noted that the separator in the drop-down menu in Figure 3 distinguishes between answers with different meanings; answers that are not separated by a line are synonyms.

The next question in the sentence, called the Immediacy option, is concerned with whether or not other events may intervene between occurrences of *button-push* and occurrences of *door-close*. As with the Pre-arity option, the developer can answer this question by opening the drop-down menu labeled "Immediacy" and selecting the appropriate choice. The third drop-down menu in the sentenced, labeled "lead to," is simply a list of synonymous ways to express the causal relationship between *button-push* and *door-close*. Changing the wording by selecting a synonym does not change the meaning of the property. The final question in the sentence, called "Post-arity," is concerned with how many occurrences of *door-close* are allowed to occur after the first occurrence of *button-push*. The developer can answer this question by opening the drop-down menu labeled "Post-arity" and selecting the desired choice.

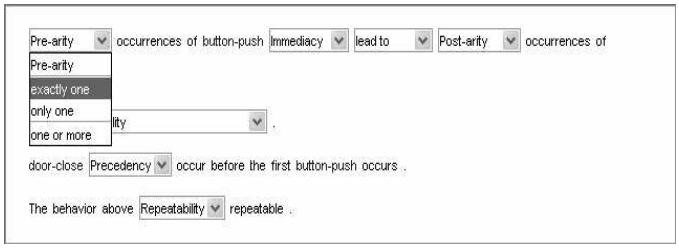

Figure 3: Answering Pre-arity in the DNL Template

Up to this point, the discussion has only been concerned with using the DNL template to edit the property, but the PROPEL tool reflects the changes in the FSA representation as well. As is shown in Figure 4, the FSA representation of the property is a state machine that has several optional components. Here, the leftmost state is an optionally-accepting start state, denoted as a state with a dashed inner concentric

circle. The developer can make this state accepting or non-accepting. If the developer decides that the start state should be accepting, it means that *button-push* is not required to occur; the property would be satisfied if that event never occurred. If the developer decides that the start state should be non-accepting, it means that *button-push* is required to occur at least once in the program execution. Figure 4 shows how the developer can answer this Nullity question by means of a three-choice ("accepting," "not accepting," or "undecided"), contextual menu for the start state: the developer can select the choice that is desired.

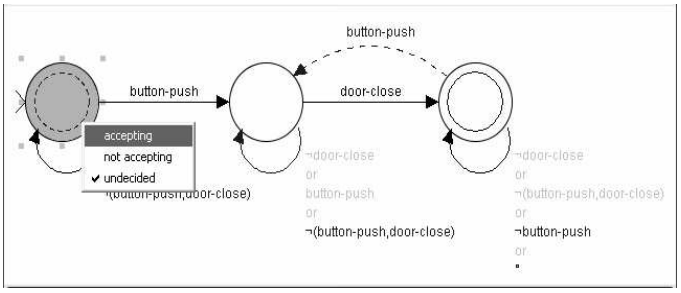

Figure 5: Answering Nullity in the FSA Template

In this figure, the transition going from the rightmost state to the middle state is an optional transition, indicated by a dashed line instead of a solid line. The developer could keep the optional transition there or remove it. If the developer decides that this transition should exist, it would mean that the behavior is repeatable; that is, further occurrences of *button-push* cause further occurrences of *door-close* to happen. If the developer decides that this transition should be removed, it would mean that further occurrences of *button-push* do not require subsequent occurrences of *door-close* to happen. Similar to the optionally-accepting start state, in Propel the developer can answer this Repeatability question by means of a three-choice ("exists," "does not exist," or "undecided"), contextual menu for the optional transition.

The final option in the Response property template, called "Precedency," is concerned with whether or not *door-close* is allowed to occur before the first occurrence of *button-push*. The developer can decide this question by selecting only one of the items in the multi-label on the start state's self-loop to be the label on that transition. A multi-label is denoted by a list of alternative sets of labels, called "multi-label items," each set separated by the word "or". If the developer decides that the label on the start state's self-loop should be ¬(*button-push*,*door-close*), it means that *door-close* is not allowed to occur until *button-push* has occurred at least once, since the "¬" operator provides a shorthand notation to indicate the complement of the given set of events with respect to the property alphabet. If the developer decides that the label on that transition should be ¬*button-push*, it means that *door-close* is allowed to occur before the first occurrence of *button-push*.

Throughout the editing of the templates, the PROPEL tool

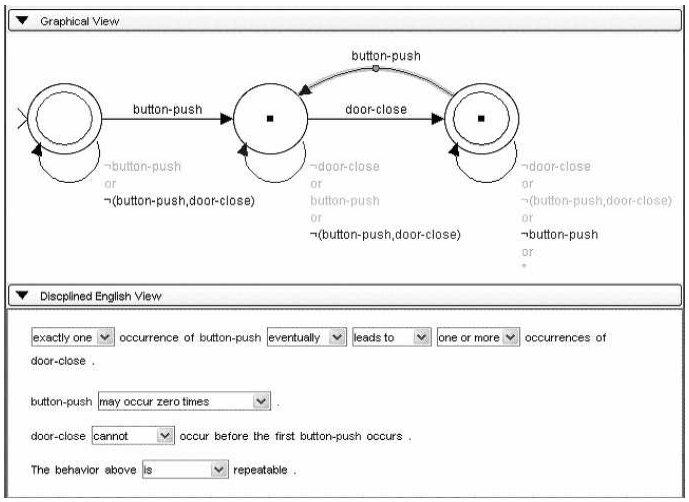

Figure 4: An Instantiated Response Property

reflects the changes in both the DNL and FSA representations. Figure 5 shows the finished version of the property in both representations. After fully-instantiating the template, the FSA template is resolved to an FSA property and the DNL template is resolved to a completed natural language paragraph. Any option in a property 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 developers ask questions about the requirements and to elucidate the meaning of a property. The developer 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.

A similar process for editing the property's scope is supported as well. The process of deciding which scope is appropriate can also be represented as a decision tree. In addition to this representation, the PROPEL tool provides a scope DNL paragraph and an associated graphical timeline representation as part of its description of the property. As with the behaviors, any change to one of these scope representations is reflected in all of these representations.

V. RELATED WORK

The PROPEL approach described in this paper builds directly on the property patterns [9]. 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. These patterns form the basis of the extensible specification language in the Bandera system [4, 5], 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 [6, 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 [12, 13] offers annotated templates to guide non-expert users, and the Cico/Circe [1] tool includes suggested phrases for expressing relationships between artifacts. The FormalCheck [11] finite-state verification tool uses templates to formulate the properties to be checked. The PROPEL approach is unique in that it incorporates templates in both natural language and a formal notation to specify properties.

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, supports similar analyses [15]. These approaches are general formalisms for expressing primarily state-based requirements, while the PROPEL approach focuses more on helping to elucidate the options associated with event-based requirements.

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 developer to move back and forth between them in order to help make the formal specifications more understandable and accessible, but does not attempt to understand natural language, even in restricted domains.

VI. 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, as natural language phrases, and as a series of questions structured as a decision tree. We hypothesize that this approach will help developers elucidate the precise meaning of the properties that they are expressing. 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 and explore the solution space more fully by re-examining the interaction between scopes and behaviors, such as considering options for scopes and using the decision tree structure independent of the pattern system. We also plan to develop translations into other precise formalisms for the purpose of integration with testing and verification tools, and we plan to improve the NL representation to increase developers' ease with the approach. Most importantly, we want to evaluate the PROPEL approach and discover ways to improve it. Although we have applied this approach to several properties and have been pleased with the results, we need to undertake a careful and extensive evaluation.

VII. ACKNOWLEDGEMENTS

This research was partially supported by the U.S. Department of Defense/Army Research Laboratory and the U.S. Army Research Office under Agreement DAAD190110564 and grant No. DAAD19-03-1-0133 and by the National Science Foundation under Grant CCR-0205575. 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 U. S. Army, the U.S. Dept. of Defense, the U.S. Government, or the National Science Foundation.

VIII. REFERENCES

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

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

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

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, R. Zheng, and H. Zheng, "Bandera: Extracting Finite-state Models from Java Source Code," presented at 22nd International Conference on Software Engineering, Limerick, Ireland, 2000, pp. 439-448.

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

[6] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration," presented at Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, CA, 1996, pp. 179-190.

[7] M. B. Dwyer and L. A. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs," presented at Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, New Orleans, LA, 1994, pp. 62-75.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property Specification Patterns for Finite-state Verification," presented at Second Workshop on

Formal Methods in Software Practice, Clearwater Beach, FL, 1998, pp. 7-15.

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

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

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

[12] N. E. Fuchs and R. Schwitter, "Attempto Controlled English (ACE)," presented at First International Workshop on Controlled Language Applications, 1996.

[13] N. E. Fuchs, U. Schwertel, and R. Schwitter, "Attempto Controlled English -- Not Just Another Logic Specification Language," presented at Eighth International Workshop on Logic-Based Program Synthesis and Transformation, 1998, pp. 1-20.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[15] M. P. E. Heimdahl and N. G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Transactions on Software Engineering*, vol. 22, 1996, pp. 363-377.

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

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

[18] G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, 1997, pp. 279-294.

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

[20] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks," presented at Third International Conference on Requirements Engineering, 1997.

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

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

[23] D. O. Paun and M. Chechik, "Events in Linear-Time Properties," presented at Fourth International Conference on Requirements Engineering, Limerick, Ireland, 1999.