Facilitating the Construction of Specification Pattern-based Properties*

Sascha Konrad and Betty H.C. Cheng†
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, Michigan 48824 USA
{konradsa,chengb}@cse.msu.edu

Abstract

Formal specification languages are often perceived as difficult to use by practitioners, and are therefore rarely used in industrial software development practices. Numerous researchers have developed specification pattern systems to facilitate the construction of formal specifications of system properties. Feedback indicates that these patterns are considered helpful, but many practitioners prefer capturing properties using informal notations, such as natural language, instead of formal specification languages. This paper describes a project that addresses this technology gap. First, we introduce a stepwise process for deriving and instantiating system properties in terms of their natural language representations. The key components of this process are structured natural language grammars and specification pattern systems. Second, we describe SPIDER*, a prototype implementation of a tool suite supporting this specification process. We illustrate the use of our approach with a description of a stepwise construction process of property specifications of a real-world automotive embedded system using* SPIDER*.*

1. Introduction

Temporal logics, in particular, real-time temporal logics, are often perceived by practitioners as difficult to understand and apply. Feedback from industrial collaborators indicates that, typically, only developers with extensive training in formal methods are inclined to make use of temporal logics. Thus, temporal logics and other precise specification languages are rarely used in the development of software systems [13], even though they can be used to capture properties formally in a precise and unambiguous manner and can be automatically analyzed with model checkers [4, 16, 18, 29], theorem provers [27], and other formal analysis tools [7]. To promote the use of formal specification techniques, we previously developed real-time specification patterns [19] to be used in combination with the already established qualitative specification patterns by Dwyer *et al.* [8]. These patterns are intended to capture critical system properties and to be used with previously developed object analysis patterns [20]. In order to further enhance the accessability of these specification patterns and their analysis tools, this paper introduces a syntax-guided approach to deriving and instantiating qualitative and real-time specification patterns in terms of their natural language representations. We present SPIDER (**S**pecification **P**attern **I**nstantiation and **D**erivation **E**nvi**R**onment), which provides a graphical environment to a tool suite supporting the derivation and instantiation of specification patterns in terms of their natural language representation.

Numerous techniques have been developed to translate requirements specified in natural language into formal specifications. Most of these approaches [3, 11, 15, 25, 31] use some type of parsing and natural language processing to construct formal specifications that can then be analyzed. Often, these informal specifications are initially mapped to an intermediate representation, at which point context dependencies and ambiguities are resolved. The result is then further refined into the targeted formal specification language(s). A common problem with these approaches is the limited ability to customize the specification style for specific domains, since the same natural language specification may potentially vary in meaning in different domains. In addition, for specification techniques that are based on standard linguistic approaches, it is difficult to ensure that a free-form user-constructed natural language sentence can be translated into a legal formal specification [30].

*This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, CDA-9700732, CCR-9901017, Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, Eaton Corporation, Siemens Corporate Research, and in cooperation with Siemens Automotive, Detroit Diesel Corporation, and General Dynamics Land Systems.

†Please contact this author for all correspondences.

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

We offer two key insights to address these problems. First, by using a grammar-driven derivation-based approach to construct natural language representations of property specifications, it is only possible to construct natural language specifications that have corresponding formal specifications. Second, we base the grammar on a specification pattern system coupled with corresponding natural language representations. This approach enables us to customize the specification system according to commonly-occurring properties, natural language vocabulary, and specification style of the domain. In our approach, domain experts work closely with formal methods experts in constructing the specification grammar. Specifically, the domain experts use natural language to capture appropriate properties identified for a given domain. The formal methods experts decide how to best represent these properties in terms of appropriate target specification languages. The resulting specifications can then be organized into a specification pattern system with an accompanying structured natural language grammar.

This paper describes three major contributions. First, we introduce a stepwise approach to deriving natural language specifications that can be mapped to a repository of temporal logic formulas based on qualitative and real-time specification patterns. We illustrate the derivation process using a structured English grammar that we developed for specifying qualitative specification patterns [8] and real-time specification patterns [19] in terms of an informal representation. Second, SPIDER supports a model-based instantiation of a natural language property, where a property is instantiated with information automatically extracted from a formal system model. This formal system model can be obtained from the specifications of formal analysis environments, such as a previously developed UML formalization framework [24]. SPIDER is customizable for different domain-specific natural language vocabularies, specification pattern systems, and analysis tools. For example, the vocabulary and natural language specification style to capture a cause-effect property for the embedded systems domain may be quite different than that used for a web service application. As such, the mappings from the structured natural language grammar to the specification patterns should reflect the appropriate intent. Finally, SPIDER offers transparency to the formal presentation of patterns and properties. In this paper, we illustrate the applicability of our approach to specifying the real-time requirements of an electronically controlled steering system obtained from one of our industrial collaborators.

Overall, our approach combines the completeness of a pattern system for specifying qualitative and real-time properties with the accessibility of a natural language representation. The remainder of the paper is organized as follows: Section 2 reviews object analysis patterns and specification patterns. Section 3 describes our specification construction and analysis process, a structured natural language grammar, and our SPIDER tool suite. Section 4 illustrates the approach with an example derivation and instantiation of critical properties of an electronically controlled power-assisted steering system. Section 5 examines related work. Finally, in Section 6 we present conclusions and discuss future work.

2. Background

In this section, we overview object analysis patterns for embedded systems and the specification patterns by Dwyer *et al.* [8].

2.1. Object Analysis Patterns

Previously, we investigated how an approach similar to the well-known design patterns [12], termed *object analysis patterns*, can be applied in the analysis phase of embedded system development [20]. Object analysis patterns contain structural and behavioral information that can be used by developers to quickly construct conceptual models of their systems in terms of UML diagrams. In addition, they contain a *Constraints* field with property templates specified in terms of linear-time temporal logic (LTL) [23]. These property templates are based on the specification patterns by Dwyer *et al.* [8]. We developed MINERVA, a tool suite that supports a pattern-driven approach to creating and analyzing UML models that uses *Hydra*, a previously developed UML formalization framework [24] to automatically generate formal specifications from UML diagrams. The work described in this paper leverages MINERVA and *Hydra*, while focusing on facilitating the construction of formal property specifications based on the specification templates in the *Constraints* field of our object analysis patterns.

2.2. Specification Patterns

Dwyer *et al.* [8] developed several patterns applicable to software properties specified in different formalisms, such as LTL [23], computational tree logic (CTL) [5], graphical interval logic (GIL) [32], and quantified regular expressions (QRE) [28]. Specification patterns are categorized into two major groups: *occurrence patterns* and *order patterns*. While a given specification pattern may have several *scopes* of applicability (*e.g.*, globally, before an event/state occurs, after an event/state occurs), the original specification patterns do not include timing information. Therefore, we refer to the specification patterns by Dwyer *et al.* as *qualitative specification patterns* as they specify qualitative properties that are not amenable to quantitative reasoning about time.

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

In our preliminary work with requirements of embedded systems, it became clear that many of the requirements were often timing-based, which could not be specified in terms of Dwyer *et al.*'s specification patterns [8] in the *Constraints* field of our object analysis patterns [20]. Therefore, we have identified a number of real-time specification patterns [19] complementary to the qualitative specification patterns. These patterns are specified in terms of metric temporal logic (MTL) [2, 21], timed computational tree logic (TCTL) [1], and real-time graphical interval logic (RTGIL) [27], all real-time extensions to logics used for the qualitative specification patterns. The real-time specification patterns can be used to specify properties about the duration that a boolean proposition holds, the periodic occurrence of a satisfied boolean proposition, and the timing-dependent order in which boolean propositions hold.

3. Specification Derivation and Instantiation

In this section we introduce our specification derivation and instantiation process and describe the corresponding tool support.

3.1. Example Scenario

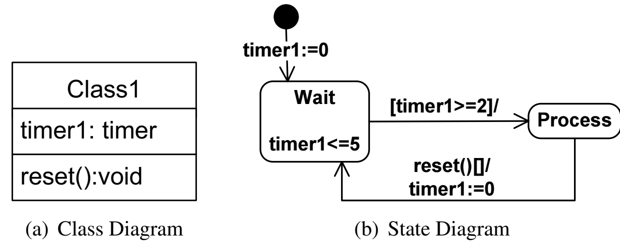
The following gives an example scenario that illustrates our specification approach: Figure 1 contains UML models that will be used as a running example throughout this section, comprising the UML class Class1 in Figure 1(a) with the corresponding state diagram in Figure 1(b). We use timed automata-like constructs to capture timing constraints in UML diagrams (please refer to [18] for details). For the remainder of the paper, UML elements adhere to the following style conventions: classes and states are named in a san serif font, method names and messages are denoted in *italics*, and attribute names are given in `typewriter` font. Assume a user wants to verify the following property:

"Whenever Class1 is in state Wait, it will enter the state Process within 5 time units." (1)

The requirement captures a cause-effect relation, in which "Class1 is in state Wait" causes Class1 to "enter the state Process" with a real-time constraint of "within 5 time units".

Using our specification approach, the user follows a stepwise process to *derive*[1] a structured natural language sentence capturing the requirement. This sentence can then be *instantiated* with UML model elements (from Figure 1) and then be mapped to a temporal logic formula in the specification pattern system. For example, the natural language requirement in Expression (1) can be systematically refined

[1]For this paper's purposes, the word "deriving" refers to the process of constructing a natural language sentence in a stepwise fashion that satisfies the structured natural language grammar.



(a) Class Diagram (b) State Diagram

Figure 1. Example UML models

into the following MTL specification, a real-time specification pattern instance [19], which can then be used by formal analysis tools, such as the *Temporal Rover* [7] and *Hydra* [18]:

$$\Box((\texttt{in(Class1.Wait)}) \rightarrow \Diamond_{\leq 5}(\texttt{in(Class1.Process)})). \qquad (2)$$

3.2. Specification Process

We have developed a process for deriving and instantiating analyzable properties based on specification patterns. Briefly, this process comprises three steps:

1. **Derivation:** Derive a natural language sentence from structured grammar.
2. **Instantiation:** Instantiate the natural language representation with model-specific elements.
3. **Mapping:** Map the instantiated natural language sentence to the temporal logic required by the targeted formal validation and verification tool and analyze.

An important component of our process is a natural language grammar that is used to derive natural language sentences that can be mapped to formal specifications structured in terms of a specification pattern system. In this paper, we use a structured English grammar and a specification pattern system that supports both qualitative and real-time specification patterns [19]. Figure 2 contains the grammar,[2] where literal terminals are delimited by quotation marks (" "), non-literal terminals are given in a san serif font, and non-terminals are given in *italics*. The start symbol S of the grammar is `property` and the language $L(G)$ of the grammar is finite. Each sentence (or string) s with $s \in L(G)$ serves as a handle that accompanies a scoped formula of a qualitative or real-time specification pattern. Therefore, the grammar aids in understanding the meaning of a property without needing to analyze the temporal logic representation. Our structured English grammar is similar in spirit to the structured English grammar for clocked computational tree logic (CCTL) by Flake *et al.* [10]. One major difference is that our grammar is based on a specification pattern system and therefore intended to be general

[2]The grammar shown contains only production rules referenced in this paper. For the complete grammar, please refer to [19].

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

Start	1: property	::=	*scope* ", " *specification* "."
Scope	2: scope	::=	"Globally" | "Before " R | "After " Q | "Between " Q " and " R | "After " Q " until " R
General	3: specification	::=	*qualitativeType* | *realtimeType*
Qualitative	…		…
	10: orderCategory	::=	"it is always the case that if " P " holds" (*precedencePattern* | *precedenceChainPattern1-2* | *precedenceChainPattern2-1* | *responsePattern* | *responseChainPattern1-2* | *responseChainPattern2-1* | *constrainedChainPattern1-2*)
	…		…
	14: responsePattern	::=	", then " S " eventually holds"
	…		…
Real-time	18: realtimeType	::=	"it is always the case that " (*durationCategory* | *periodicCategory* | *realtimeOrderCategory*)
	…		…
	22: periodicCategory	::=	P " holds " *boundedRecurrencePattern*
	23: boundedRecurrencePattern	::=	"at least every " c " time unit(s)"
	…		…

Figure 2. Structured English grammar excerpt

enough to support translations of untimed and timed properties to multiple temporal logics. Specifically, our grammar supports specifications using LTL, CTL, and GIL for Dwyer *et al.*'s specification patterns [8] and MTL, TCTL, and RTGIL for our real-time specification patterns [19].

The structured English grammar is organized according to our classification of qualitative and real-time specification patterns (please refer to [19] for details on the classification), in which we group specification patterns according to categories (*qualitative* or *real-time*) and types (*duration*, *periodic*, or *real-time order* for real-time properties, and *occurrence* or *order* for qualitative properties). In general, there are four major choices to be made when constructing a natural language representation of a property with respect to this grammar. The process for using our grammar is as follows: Initially, the users determine (1) the scope of the property to be specified (*Globally*, *Before*, *After*, *Between*, or *After-until*), followed by (2) the type and then (3) the category of the property to be specified. The final structured English sentence is constructed by (4) choosing the corresponding specification pattern. In the grammar, "precede" and "succeed" denote strict past and future, respectively, while "held previously" and "hold eventually" denotes non-strict past and future, respectively. For example, "S eventually holds" is satisfied if S holds in the current state, while this is not the case with "S succeeds".

3.3. Tool Support

Figure 3 gives a data flow diagram overviewing SPIDER, where bubbles represent processes, arrows denote data flow, two parallel lines depict data stores, and external entities are represented by rectangles. Initially, *Formal methods experts* and *Domain experts* collaborate to create a domain-specific collection of formal specifications and associated natural language representations. Using the *Pattern System Manager*, they then construct a *Structured natural language grammar with descriptors* for the natural language representations and accompanying *Formal specification mapping* files that describe how to map these natural language representations to formal specifications. The natural language grammar is then used by the *Property Deriver* to guide the *SPIDER user* in constructing a structured language sentence capturing the property to be specified. In previous investigations, we developed a *UML formalization framework*, including the *Hydra* tool, that produces formal specifications for UML diagrams [18, 24]. For the current project, the formal system model comes from *Hydra*. The *Formal Model Interpreter* automatically extracts information from the system model, which is used by the *Property Instantiator* to instantiate the structured language sentence with boolean propositions containing model-specific elements. In addition, the *Property Instantiator* invokes the *Property Analyzer* corresponding to the targeted analysis tool. At this point, the *Property Analyzer* maps the instantiated natural language sentence to the corresponding specification pattern instances understood by the targeted analysis tool. Finally, the *Property Analyzer* provides analysis results back to the *Property Instantiator*, which are then visually presented to the user.

SPIDER can be configured to support commonly-occurring properties, natural language vocabulary, and specification style of a domain. More specifically, SPIDER can be configured as follows:

IEEE COMPUTER SOCIETY

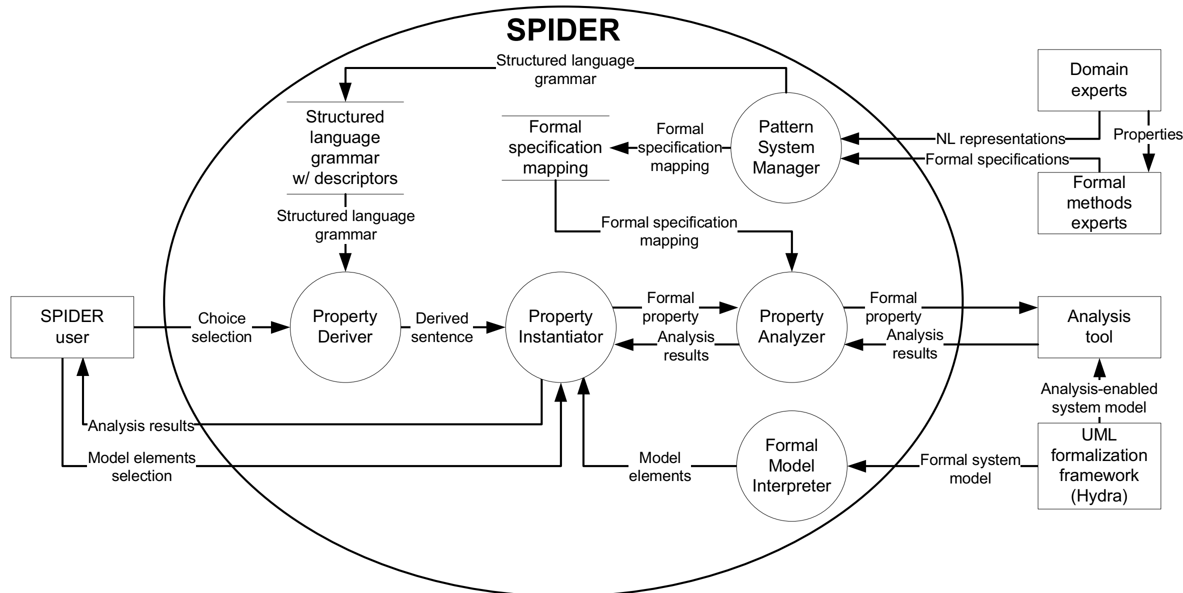


Figure 3. Data flow diagram overviewing our SPIDER tool suite

Pattern System Manager: SPIDER offers graphical support for creating and modifying structured natural language grammars. In addition, the formal specifications as well as their associations to natural language sentences can be modified. Currently, SPIDER includes a grammar and specification patterns for the specification patterns by Dwyer *et al.* [8], as well as a grammar and specification patterns for an extended specification pattern system (used in this paper) that also includes our real-time specification patterns [19].

Formal Model Interpreter: SPIDER uses an abstract, model-independent representation of a formal system model. This representation is populated by the *Formal Model Interpreter* tailored for the specific formal analysis tool. Presently, SPIDER supports the extraction of information from formal models created by *Hydra* [24], as well as Promela models for the model checker Spin [16].

Property Analyzer: The formal analysis tool-specific *Property Analyzer* is responsible for the steps required to invoke the formal analysis tool. The *Property Analyzer* first maps the natural language specification to the specification logic understood by the targeted formal analysis tool. During this mapping, the model-specific elements in the natural language specification are also mapped to a representation expected by the analysis tool. Finally, the *Property Analyzer* invokes the formal analysis tool with parameters required to verify the system model under consideration. Currently, SPIDER supports analysis using the Spin model checker [16] and support for Kronos [4] is being developed.

The main elements of SPIDER, the *Pattern System Manager*, the *Property Deriver*, and the *Property Instantiator* with the *Formal Model Interpreter* and the *Property Analyzer* are now described in more detail:

Pattern System Manager. The *Pattern System Manager* is intended to be used by the domain experts and formal methods experts as an administrative tool that configures SPIDER according to a specification pattern system. It aids in the construction and management of specification pattern systems with their associated structured language grammars. Specification patterns can be associated with sentences automatically extracted from the language of the grammar and these associations are stored in mapping files. When the natural language grammar or the specification pattern system is modified, the *Pattern System Manager* preserves as many mappings as possible, prompting the user to reassign elements that have no mappings associated. The grammar is captured in Extended Backus-Naur Form (EBNF) and internally translated to a BNF representation. The use of EBNF leads to a higher level of abstraction and aids in the understandability and readability of the grammar, while BNF is easier to process when providing user guidance, due to simplicity. For grammar rules containing choices (*e.g.*, *scope* and *specification* in Figure 2), additional descriptors are included. These descriptors comprise two parts: An abbreviated name of the choice and a textual explanation of each choice. This information is used in the derivation process to provide guidance and feedback to the user when making a choice in the derivation process.

The *Pattern System Manager* is also used by the formal methods experts to specify the mappings between the sen-

IEEE COMPUTER SOCIETY

tences generated from the natural language grammar and elements from the specification pattern system.

Property Deriver. Once SPIDER has been instantiated with a natural language grammar, mappings to a specification pattern system, and appropriate analysis tools, the *Property Deriver* guides the user through the step-by-step derivation of a natural language representation of the property to be specified. Non-terminals are highlighted in the sentence that is being derived and the user resolves these non-terminals with applicable production rules. The *Property Deriver* assists the user in making specification choices by offering descriptive information about the consequences of each choice. Each time the user highlights a particular choice, the *Property Deriver* highlights corresponding descriptors. In addition, the *Property Deriver* gives a preview of selecting a particular choice for the sentence being derived.

Using the *Property Deriver* for our running example, the user would derive the following structured English sentence from Expression (1):

> **"Globally, it is always the case that if P holds, then S holds within c time units."** (3)

This sentence captures a real-time response property, where P, S, and c are placeholder variables to be instantiated later with formal system model elements.

Figure 4 shows a screen capture of the *Property Deriver*. The *Sentence derivation* field shows the current sentence obtained from the derivation process. The *Choices* field contains the specific choices at this step, and the *Description* field gives descriptions for each choice. The *Derivation history* contains a tree representation of the non-terminals resolved thus far, and it can also be used to undo edits.

Property Instantiator, Formal Model Interpreter, and Property Analyzer. Figure 5 shows a screen capture of the *Property Instantiator*. The *Sentence instantiation* field contains elements that are used to instantiate the derived natural language sentence. The *Property Instantiator* extracts model-specific information from formal system models (displayed in the *Current model elements* fields). To accomplish this task, the *Property Instantiator* potentially uses several *Formal Model Interpreters*, each of which has the ability to read a certain input format of a formal model. For example, in this paper the *Formal Model Interpreter* extracts information from the *Hydra Intermediate Language* (HIL) models of our *Hydra* UML formalization framework [24] to obtain *state*, *signal*, and *variable* names for each *class*. These elements can be inserted into boolean propositions to be used in the placeholder variables of a property.

For example, for the class and state diagrams in Figure 1, the following information is extracted:

> **State names:** Wait, Process
> **Signal names:** *reset*
> **Variable names:** `timer`

A boolean proposition field in the *Property Instantiator* helps the user construct boolean expressions using elements from the formal system model. For our running example in Expression (3), the condition for P is instantiated with `(in(Class1.Wait))` (meaning that Class1 is in state Wait), the condition for S is instantiated with `(in(Class1.Process))` (meaning that Class1 is in state Process), and the value of c is set to `5` time units.

The structured English result from this instantiation step is as follows:

> **"Globally, it is always the case that if `(in(Class1.Wait))` holds, then `(in(Class1.Process))` holds within 5 time units."** (4)

The *Property Instantiator* uses a *Property Analyzer* component to translate the instantiated natural language sentence into the formal specification language of the targeted formal specification tool and to invoke the analysis. For our real-time specification patterns [19], the instantiated natural language sentence in Expression (4) is automatically mapped to the temporal logic representation in Expression (2). The *Property Analyzer* returns the analysis results in the *Analysis results* field and sets the color of the traffic light, all within the *Property Instantiator* window. A red color indicates that the property was violated and a counter example is returned; a green color indicates that the property holds for the selected model; and a yellow color indicates that problems occurred during the analysis process that prohibited the successful verification of the property. Example problems include exceeding the available system memory for storing the states of the model during an exhaustive state space exploration. By offering the ability to plug-in additional *Formal Model Interpreter* and *Property Analyzer* components, SPIDER is configurable to support numerous analysis tools beyond the ones explicitly mentioned in this paper.

4. Illustrative Example

We illustrate this approach with an example specification derivation and instantiation on an electronically controlled steering (ECS) system [36] obtained from one of our industrial collaborators, Siemens Automotive. The ECS system is intended to supplement the benefits provided by traditional hydraulic power steering using an electric-motor

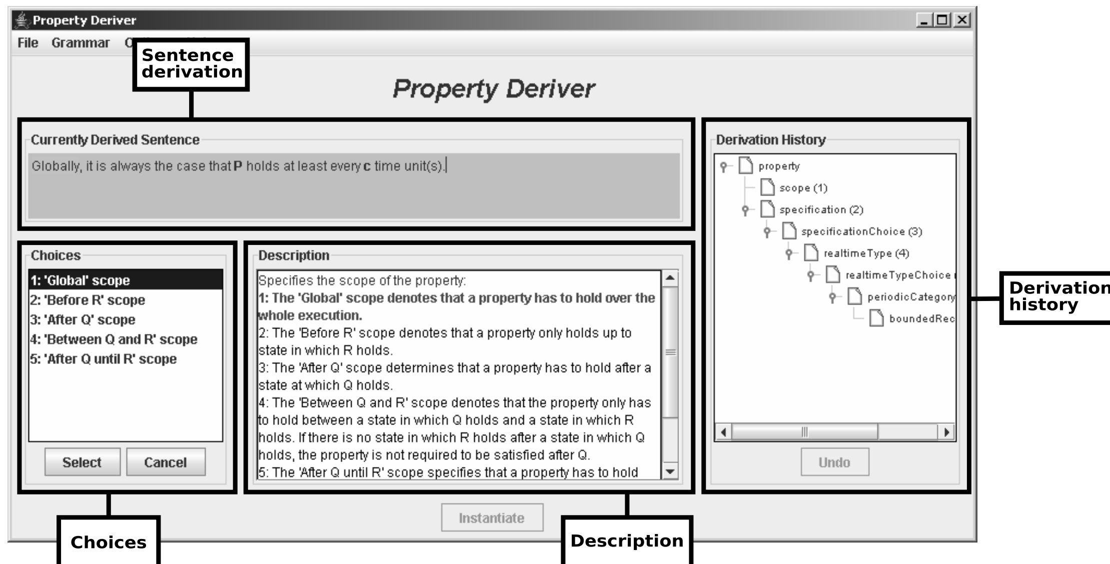


Figure 4. Annotated screen capture of the *Property Deriver*

power assist mechanism to provide responsive power steering.

4.1. Using SPIDER

We demonstrate the use of SPIDER with example instantiations of both qualitative and real-time properties of the ECS system. For brevity, global scope is used for all properties.

Property 1

Requirement: "If a failure occurs, the malfunction indicator light should be illuminated."

This requirement does not contain any timing information, therefore the type is *qualitative*. Clearly, the property should be in the *order* category, since the requirement specifies that *after* a state in which a failure occurs, the system should transition to a state where the malfunction indicator light is illuminated. Finally, the user selects the *Response Specification Pattern* from Dwyer *et al.*'s patterns [8], since the requirement to be specified is a cause-effect relation and involves two conditions. As a result, the user obtains the following structured English sentence upon completion of the derivation step:

"Globally, it is always the case that if P holds, then S eventually holds." (5)
(Grammar 1, 2, 3, 10, 14)

Next, the *Property Instantiator* is used to replace P and S with boolean propositions describing the appropriate states. SPIDER displays all the classes, with accompanying signal, state, and variable names, from the system model. In this case, we are interested in the variables of the FaultHandler and MalfunctionIndicatorLight classes. Assuming that the boolean proposition `(FaultHandler.NoOfFailures≠0)` indicates the occurrence of failures and `(MalfunctionIndicatorLight.Status=1)` denotes that the malfunction indicator light is activated, the following structured English sentence is obtained after the instantiation step:

"Globally, it is always the case that if `(FaultHandler.NoOfFailures≠0)` holds, then `(MalfunctionIndicatorLight.Status=1)` eventually holds." (6)

This instantiated sentence can then be mapped to a temporal logic representation. The following temporal logic property, automatically generated by SPIDER, denotes the LTL representation of the property captured in natural language in Expression (6), which can be analyzed by the model checker Spin [16]:

$$\Box(\texttt{(FaultHandler.NoOfFailures}\neq\texttt{0)} \rightarrow \Diamond\texttt{(MalfunctionIndicatorLight.Status=1)}). \quad (7)$$

Property 2

Requirement: "Once every second, a fault status report must be sent over a controller area network (CAN) communication link."

Property 2 clearly contains timing information (*i.e.*, "Once every second"), therefore the user selects the *real-*

IEEE COMPUTER SOCIETY

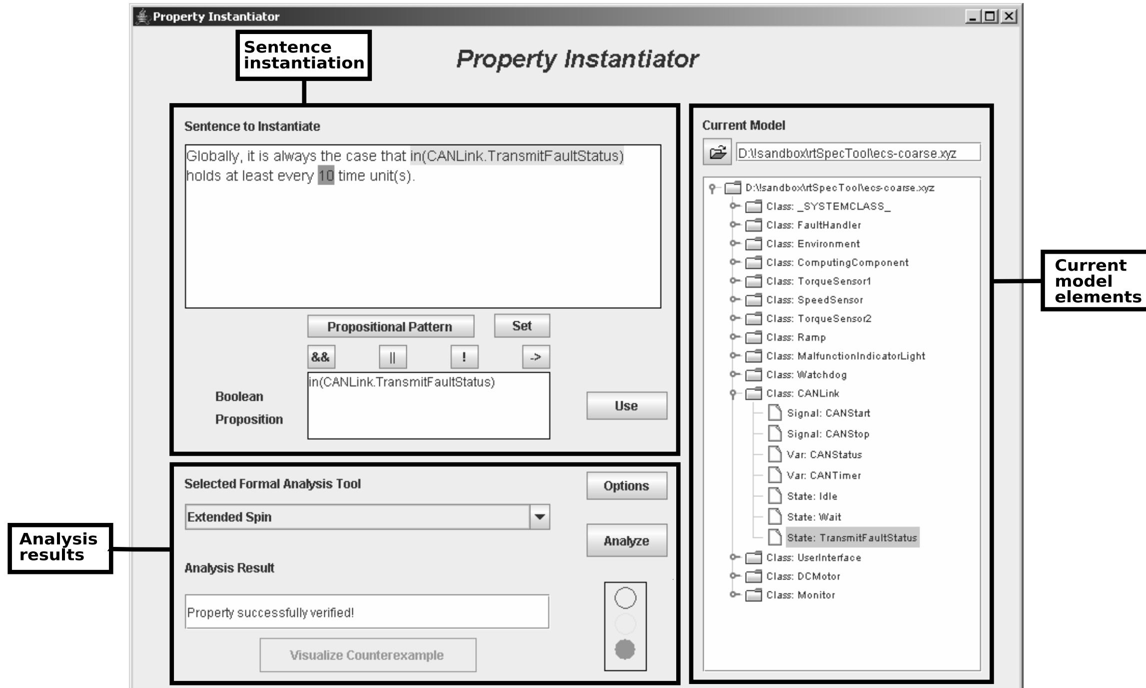


Figure 5. Annotated screen capture of the *Property Instantiator*

time type when deriving the corresponding property. Next, the user indicates whether the property under consideration falls in the *duration*, *periodic*, or *real-time* category. Since this property denotes a recurring event, the property falls in the *periodic* category, and then the user selects the *Bounded Recurrence Specification Pattern* [19]. Therefore, the derivation step yields the following structured English sentence:

> **"Globally, it is always the case that P holds at least every c time unit(s)."** (8)
> (Grammar 1, 2, 3, 18, 22, 23)

Let proposition `(in(CANLink.TransmitFault-Status))` denote that the CANLink is in state Transmit-FaultStatus, in which it transmits the current fault status of the system over the CAN link.[3] In addition, we assume that the timing granularity of the model is equal to 100 milliseconds, which means that 10 time units capture one second. Therefore, `c` is replaced with 10, denoting 10 time units. This instantiation leads to the following instantiated structured English sentence:

> **"Globally, it is always the case that `(in(CAN-Link.TransmitFaultStatus))` holds at least every `10` time unit(s)."** (9)

[3]The Controller Area Network (CAN) is a high-integrity serial data communications bus for real-time applications [34].

which SPIDER maps to the MTL formula:

$$\Box(\Diamond_{\leq 10}(in(CANLink.TransmitFaultStatus)))). \quad (10)$$

This MTL formula can be used by our timing-extended UML formalization framework [18] to check for adherence of the formalized UML system model to the requirement under consideration.

5. Related Work

Numerous approaches [3, 11, 15, 25, 31] construct formal specifications in different forms (such as temporal logics, OO-based representations, Prolog specifications), from natural language to support a variety of tasks, ranging from completeness and consistency checking to formal validation and verification. While these approaches allow the use of moderately restricted natural language (a completely unrestricted language is considered undesirable for practical and technical reasons [31]), this type of extraction is a more ambitious goal than our approach using syntax-guided derivation and model-based instantiation, since it requires advanced natural language processing approaches and techniques to deal with imprecision and ambiguities inherent to natural language specifications.

Fantechi *et al.* [9] use natural language constructs that map directly into ACTL, an action-based variant of CTL. User input is used to resolve ambiguities that might be encountered in the input phrase. While this approach is di-

IEEE COMPUTER SOCIETY

rectly related to ours, it restricts the source language according to the structure of the targeted temporal logic [31]. While our approach also restricts the source language, this restriction is not due to the structure of the targeted temporal logic(s), but, instead, is due to the previously created natural languages representations. As such, our grammar can support multiple target languages because a specification pattern system may include mappings to multiple formal specification languages.

In summary, none of the aforementioned approaches combines the completeness of a patterns system, the support for real-time properties, amenability for formal validation and verification with a wide variety of formal validation and verification tools, and the accessibility of a natural language representation in any natural language subset for which a context-free, non-recursive grammar without repetitions can be constructed.

Smith *et al.* developed Propel [35], where they extended the specification patterns by Dwyer *et al.* [8] to address important and subtle aspects about a property, such as what happens in a cause-effect relation if the cause recurs before the effect has occurred. These extended specification patterns are specified in terms of finite-state automata instead of temporal logic formulae. Smith *et al.* also offer disciplined natural language templates that help a specifier to precisely capture a property in natural language. The natural language templates are comparable to the natural language-based representations that can be created using our structured English grammar. Similar to our syntax-guided derivation process, Propel also offers decision tree templates to aid the user in deciding which property template is suited for specifying the intended property. Differing from our approach, Propel does not offer support for including real-time information and does currently not include (customizable) support for instantiating a property with information automatically extracted from a formal model and analyzing the formal model for adherence to the specified property.

Mondragon *et al.* developed a tool called Prospec [26] for the specification of properties based on Dwyer *et al.*'s specification patterns. The tool offers assistance in the specification process and extends the specification pattern system by Dwyer *et al.* with compositional patterns. Differing from our tool suite, they do not include support for natural language representations or real-time information.

Other tools related to SPIDER are general-purpose syntax-directed editors without inherent support for formal analysis, such as the Synthesizer Generator [33]. In addition, tools like the IFADIS toolkit [22] and the MT toolset [6] offer temporal logic templates to specify properties to be checked, but do not incorporate natural language support.

6. Conclusions

This paper described our configurable process for derivation and instantiation of analyzable natural language properties. We have implemented this approach in our SPIDER tool suite by supporting both existing qualitative and real-time specification pattern systems. We illustrated the derivation and instantiation process on an embedded system from the automotive industry. Combined with our UML formalization framework, users are able to specify system models in terms of UML and analyze these models using the natural language representation of specification patterns obtained from the derivation and instantiation process supported by SPIDER.

We acknowledge that the stepwise, specification-facilitating features, while helpful for the novice user, might be too constraining for users with advanced knowledge in formal specification and analysis. This problem is commonly encountered in syntax-directed editing approaches [17] and we plan to investigate techniques to mitigate these problems, such as the use of multiple views and different levels of assistance for the derivation and instantiation tasks. In addition, we also plan on investigating how to assist users in constructing the natural language grammar, since the grammar is pivotal for the effective application of our approach. Other directions for future work are also possible, such as integrating extensions that address property subtleties similar to the Propel work [35].

Finally, support for additional analysis tools could be integrated into SPIDER. Currently, SPIDER supports our timing-extended UML formalization framework and the model checker Spin [16], and support for Kronos [4] is being developed. Other possible extensions include the model checking tools UPPAAL [29] and Hytech [14]. Also, the usability of the tool could be enhanced according to user feedback. Additional pattern systems and natural languages grammars could be included in the tool suite and we expect that different combinations of temporal logics and application domains will lead to different sets of pattern systems and natural language grammars.

Acknowledgements
The authors gratefully acknowledge Anthony Torre for valuable comments on this work throughout the project. We also thank the faculty and students from the Software Engineering and Network Systems (SENS) Laboratory at Michigan State University, in particular Laura Campbell and Min Deng. Finally, we greatly appreciate the comments and feedback from the anonymous reviewers.

References

[1] R. Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford University, 1991.
[2] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.

IEEE COMPUTER SOCIETY

[3] V. Ambriola and V. Gervasi. Processing natural language requirements. In *Proc. of the 12th Int. Conf. on Automated Software Engineering*, pages 36–45, Lake Tahoe, NV, 1997.

[4] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th Int. Conf. on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer-Verlag, 1998.

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

[6] P. Clements, C. L. Heitmeyer, B. G. Labau, and A. T. Rose. MT: A toolset for specifying and analyzing realtime systems. In *IEEE Real-Time Systems Symposium*, December 1993.

[7] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. of the 7th Int. SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330. Springer-Verlag, 2000.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of the 21st Int. Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press, 1999.

[9] A. Fantechi, S. Gnesi, R. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisiting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4:243–263, 1994.

[10] S. Flake, W. Mueller, and J. Ruf. Structured English for model checking specification. In *Proc. of the GI-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, Frankfurt, Germany, February 2000. VDE Verlag1. (English version).

[11] N. E. Fuchs and R. Schwitter. Attempto controlled English (ACE). In *The First Int. Workshop on Controlled Language Applications (CLAW)*, Leuven, BE, March 1996.

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

[13] C. L. Heitmeyer. On the need for practical formal methods. In *FTRTFT '98: Proc. of the 5th Int. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 18–26. Springer-Verlag, 1998.

[14] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[15] A. Holt and E. Klein. A semantically-derived subset of English for hardware verification. In *Proc. 37th Annual Meeting of the Association for Computational Linguistics: Maryland, USA*, pages 451–456. Association for Computational Linguistics, 1999.

[16] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2004.

[17] A. A. Khwaja and J. E. Urban. Syntax-directed editing environments: Issues and features. In *SAC '93: Proc. of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 230–237. ACM Press, 1993.

[18] S. Konrad, L. A. Campbell, and B. H. C. Cheng. Automated analysis of timing information in UML diagrams. In *Proc. of the Nineteenth IEEE Int. Conf. on Automated Software Engineering (ASE04)*, pages 350–353, Linz, Austria, September 2004. (Poster summary).

[19] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proc. of the Int. Conf. on Software Engineering (ICSE05)*, St Louis, MO, USA, May 2005.

[20] S. Konrad, B. H. C. Cheng, and L. A. Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970–992, December 2004.

[21] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[22] K. Loer and M. D. Harrison. Towards usable and relevant model checking techniques for the analysis of dependable interactive systems. In W. Emmerich and D. Wile, editors, *Proc. 17th Int. Conf. on Automated Software Engineering*, pages 223–226. IEEE Computer Society, September 2002.

[23] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.

[24] W. E. McUmber and B. H. C. Cheng. A general framework for formalizing UML with formal languages. In *Proc. of the IEEE Int. Conf. on Software Engineering (ICSE01)*, Toronto, Canada, May 2001.

[25] J. B. Michael, V. L. Ong, and N. C. Rowe. Natural-language processing support for developing policy-governed software systems. In *Proc. of the 39th Int. Conf. on Technology for Object-Oriented Languages and Systems*, Santa Barbara, CA, 2001.

[26] O. Mondragon and A. Q. Gates. Supporting elicitation and specification of software properties through patterns and composite propositions. *Int. Journal on Software Engineering and Knowledge Engineering*, 14(1):21–41, February 2004.

[27] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology*, 6(1):31–79, 1997.

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

[29] P. Pettersson and K. G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[30] S. G. Pulman. Controlled language for knowledge representation. In *Proc. of the First Int. Workshop on Controlled Language Applications (CLAW)*, pages 233–242, 1996.

[31] R. Nelken and N. Francez. Automatic translation of natural-language system specifications into temporal logic. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of the Eighth Int. Conf. on Computer Aided Verification CAV*, volume 1102, pages 360–371, New Brunswick, NJ, USA, 1996. Springer Verlag.

[32] Y. S. Ramakrishna, P. M. Melliar-Smith, L. E. Moser, L. K. Dillon, and G. Kutty. Interval logics and their decision procedures: Part I + II. *Theoretical Computer Science*, 166;170(1-2):1–47;1–46, 1996.

[33] T. Reps and T. Teitelbaum. The synthesizer generator. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19, pages 42–48, May 1984.

[34] Robert Bosch GmbH. Controller Area Network (CAN), 2005. http://www.can.bosch.com/.

[35] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Propel: An approach supporting property elucidation. In *Proc. of the 24th Int. Conf. on Software Engineering*, pages 11–21. ACM Press, 2002.

[36] A. Torre. Project specifications for the adaptive cruise control system and the electronically controlled steering system, 2000. http://www.cse.msu.edu/~cse470/F01/Projects/.

IEEE COMPUTER SOCIETY