Automated Analysis of Natural Language Properties for UML Models[*]

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. It is well known that errors introduced early in the development process are commonly the most expensive to correct. The increasingly popular model-driven architecture (MDA) exacerbates this problem by propagating these errors automatically to design and code. This paper describes a round trip engineering process that supports the specification of a UML model using CASE tools, the analysis of specified natural language properties, and the subsequent model refinement to eliminate errors uncovered during the analysis. This process has been implemented in SPIDER, a tool suite that enables developers to specify and analyze a UML model with respect to behavioral properties specified in terms of natural language.

1 Introduction

Errors introduced early in the development process are known to have significantly higher correction costs [1]. To worsen this problem, in the increasingly popular model-driven architecture (MDA) [2], platform-independent models are transformed to platform-specific models via transformation techniques. As such, these errors are directly propagated to the platform-specific models and may also be propagated to code, thereby motivating their detection in the platform-independent models. Validating UML models according to metrics and design guidelines can be an effective means to catch structural errors [3, 4], but generally not behavioral modeling errors. Several tools for the behavioral analysis of UML models have been developed, where a user typically specifies properties in terms of formal specification languages. However, these formal specification languages often have a complex syntax and semantics and are, therefore, rarely used in practice. To ease the use of formal specification languages, we have developed a customizable process for specifying properties of formal system models in terms of natural language and formally analyzing these properties using various formal analysis tools [5].

[*] 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.

[**] Corresponding author.

J.-M. Bruel (Ed.): MoDELS 2005 Workshops, LNCS 3844, pp. 48–57, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Several other tools exists to support the design and validation of system models. Commercial tools commonly offer validation and/or animation capabilities, such as Rhapsody [6] and Rational XDE [7]. In general, these tools aid in uncovering structural errors, but are not designed for the analysis of behavioral properties of a system model. Other tools have been developed for the formal analysis of system models specified in terms of UML, such as vUML [8], Hugo [9], and Fujaba [10]. However, these tools have still not gained a widespread use in industry. One main reason is the need to use complex specification logics and/or formal analysis tools. Consequently, only users with an advanced knowledge in formal methods are inclined to use these tools for the specification and analysis of their system models.

In this paper, we present three main contributions: First, we developed a process for specifying and analyzing formal properties, where the objective is to make the formal nature of the specification and analysis process transparent to the user. As such, property templates based on specification patterns developed by Dwyer *et al.* [11] can be specified in natural language and used to analyze the system model. We implemented this process in SPIDER (**S**pecification **P**attern **I**nstantiation and **D**erivation **E**nvi**R**onment), and we show how SPIDER can be used in combination with a previously developed UML formalization framework, termed Hydra [12], for the analysis of UML models. Second, to facilitate the specification process, we provide support for instantiating the natural language property templates with information that is automatically extracted from the formal system model under consideration. Third, the process is customizable for different domain-specific natural language vocabularies and specification styles, specification pattern systems, and analysis tools.

In this work, we show how our process can be used to specify and analyze natural language properties of UML models. More specifically, our round trip engineering process is configured to read UML 1.4 [13] models[1] specified in terms of XMI 1.1 [15] and generate the formal specification language Promela for the model checker Spin [16]. Natural language properties are derived using a grammar [17] that supports the specification patterns by Dwyer *et al.* [11]. Our grammar supports the natural language representation of these specification patterns. In this paper, the grammar is used to specify linear-time temporal logic (LTL) properties [18], the property description language of the Spin model checker. The grammar can be customized according to vocabulary and specification style of a domain. For example, the vocabulary and natural language specification style to capture a cause-effect property for the embedded systems domain may be different from 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. In addition, the semantic UML mapping rules of Hydra can be customized and adapted to other domains [12]. In this paper, we use a semantic interpretation considered to be suitable for the embedded systems domain. Our approach does not require the user to know the specific syntax and semantics of the formal specification language used or the details of the analysis procedures of the targeted formal analysis tool. An analysis process can be automatically executed and the analysis results are displayed to the user in a form easy to comprehend.

[1] CASE tool support for the recently finalized UML 2.0 [14] is still limited.

Overall, we introduce a customizable process that combines the completeness of a pattern system for specifying properties of UML models with the accessibility of a natural language representation, and present a prototype implementation termed SPIDER. To validate our approach, we have applied the process and tools to several examples from industry, including an electronically controlled steering system and an adaptive light control system. The remainder of the paper is organized as follows. Section 2 describes our process and the main components of SPIDER in more detail. Section 3 examines related work. Finally, Section 4 gives concluding remarks and discusses future work.

2 Specification and Analysis

This section introduces our specification and analysis process and also overviews major SPIDER elements. Figure 1 contains a UML activity diagram overviewing the process, where the first two steps of the process are initialization steps and can be performed in any order. The shaded swimlane indicates this portion of the process performed by an administrator for domain customization purposes. The process is illustrated with a running example, the formal specification of the natural language property:

> Whenever *process()* of the Processor has been called, then eventually the Processor returns to the Idle state.

(1) Configuring the Process and Deriving a Property
In the first initialization step, a specification pattern system has to be created. A specification pattern system is a collection of properties, specified in terms of one or more formal specification languages. Each property is also specified in terms of natural language with an accompanying natural language grammar. In SPIDER, the *Pattern System Manager* (shaded portion of Figure 1) is used to create and associate formal properties to their natural language representations, or a previously created pattern system can be loaded. For this paper, the specification pattern system consists of formal properties from the specification patterns by Dwyer *et al.* [11] and a corresponding natural language grammar [17]. The specification pattern system contains several patterns applicable to software properties specified in different formalisms, such as LTL [18], computational tree logic (CTL) [19], graphical interval logic (GIL) [20], and quantified regular expressions (QRE) [21]. Specification patterns are categorized into two major groups: *occurrence patterns* and *order patterns*. Occurrence patterns are concerned with the occurrence of single states/events during the execution of a program, such as *existence* and *absence* of certain states/events. Order patterns, on the other hand, are concerned with the relative order of multiple occurrences of states/events during the execution of a program, such as *precedence* or *response* relations between states/events. The specification patterns have been found sufficient to specify most commonly occurring properties [11]. However, while the pattern system is largely reusable, the structured natural language grammar may have to be adapted to accommodate the specification style of a specific domain.

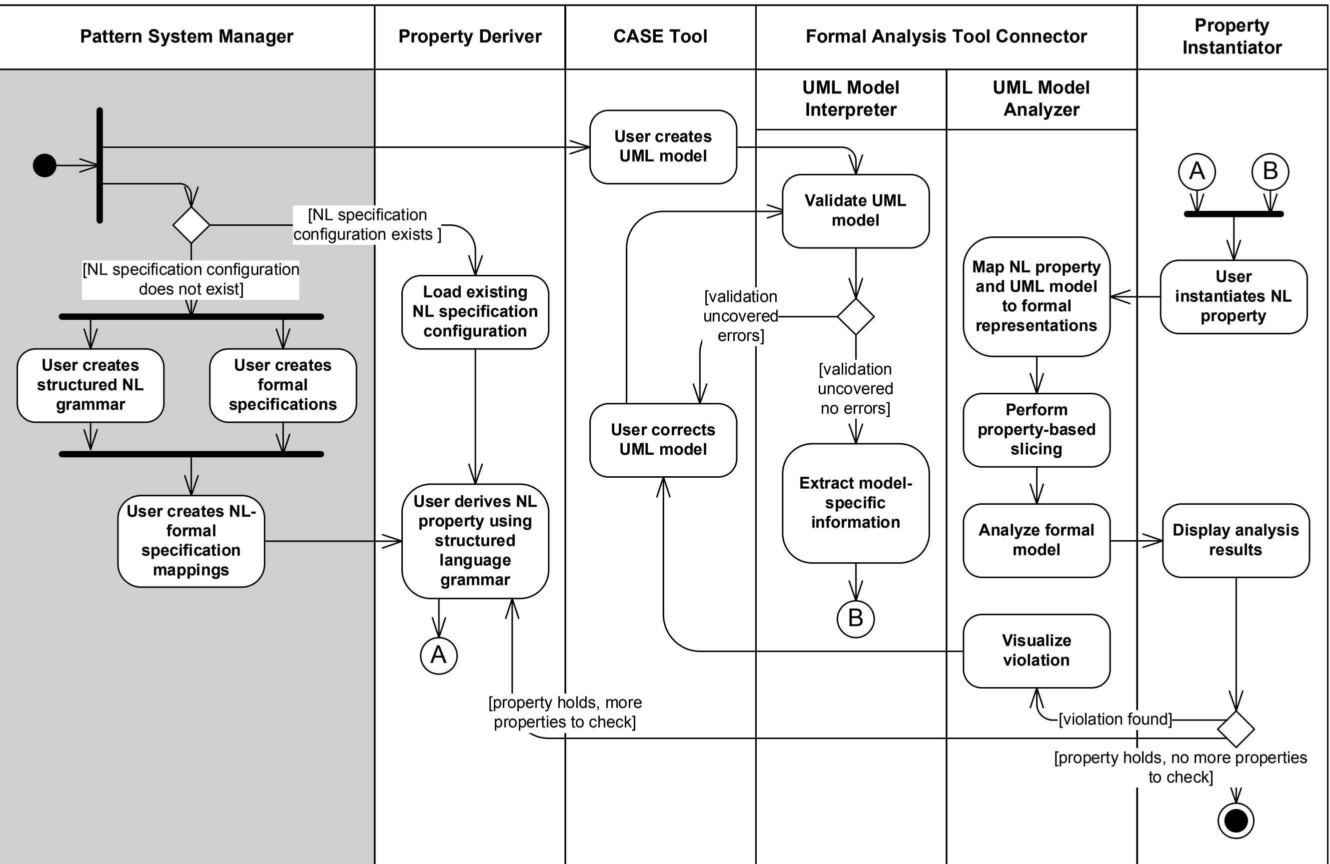


Fig. 1. UML activity diagram overviewing our specification and analysis process

The *Pattern System Manager* is intended to be used by 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 natural language grammars. Structured natural language grammars are captured in Extended Backus-Naur Form (EBNF) and internally translated into a BNF representation. For grammar rules containing choices, 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 create the mappings between the sentences generated from the natural language grammar and elements from the specification pattern system.

After the process has been instantiated with a natural language grammar and mappings to a specification pattern system, the property to be analyzed is derived. In SPIDER, the *Property Deriver* is used to guide the user in a stepwise fashion in constructing a structured natural language property template for capturing the property. Non-terminals are highlighted in the template 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 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 natural language property template being derived.

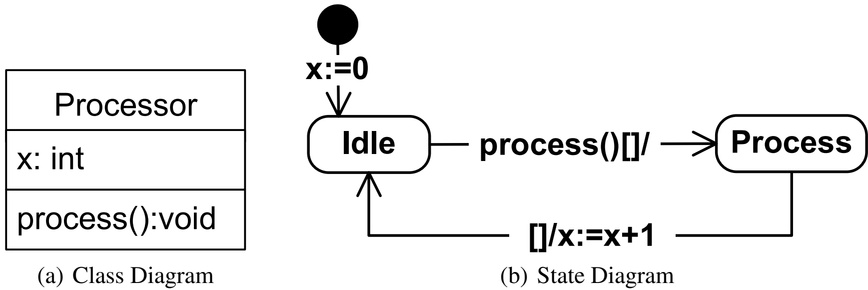
For our running example, the user needs to perform three choices during the derivation process. At first, the user needs to decide what the scope of the property is. For simplicity, we assume a global scope is selected for the property. In the next step, the user needs to choose whether the property belongs to the occurrence or order category. Since the property involves multiple occurrences, the order category is chosen. Finally, the user needs to select the appropriate specification pattern. The property describes a cause-effect relation, since the occurrence of a cause (*process()* being called) is expected to have a certain effect (the Processor returning to state Idle). Therefore, the *Response Specification Pattern* [11] is chosen. Finally, we obtain the following natural language specification template:

Globally, it is always the case that if P holds, then S eventually holds.

After the natural language template is derived, the first step ends at the connector A in Figure 1.

(2) Creating the UML Model

In the second initialization step, a UML model is created using a CASE tool. To include the model in our process instantiation, the model is exchanged with SPIDER using XMI [15]. Figure 2 shows an example UML class Processor with an associated state diagram capturing the behavior of the class. Initially, the model is validated by Hydra using static analysis techniques [22]. The model validation encompasses several checks for intra- and inter-diagram validity, such as checks for well-formedness of names and expressions, missing initial states, states without incoming or outgoing transitions, and



(a) Class Diagram (b) State Diagram

Fig. 2. Example UML model

undeclared variables, signals, or types. If errors are found during the validation that prevent a formally specified model from being generated from the UML diagrams, then the user is prompted to correct these errors before proceeding. After the model passes the validation checks, the *UML Model Interpreter* automatically extracts information about model elements from the UML model, such as the names of classes, variables, signals, and states. For example, for the UML diagram in Figure 2, the *UML Model Interpreter* extracts the following information for class Processor:

Variable name(s): `x`
Signal name(s): *process*
State name(s): Idle, Process

The *UML Model Interpreter* is part of the *Formal Analysis Tool Connector* that is used to connect SPIDER with UML tools and the Spin model checker. In general, the tool connector enables SPIDER to extract information from a system model, create formal specifications of properties in a form suitable for a particular formal analysis tool, execute the verification of a property, and analyze the output generated by a verification run of the formal analysis tool. SPIDER allows additional *Formal Analysis Tool Connector* components to be plugged in. Therefore, making it extensible to numerous analysis tools beyond the ones explicitly mentioned in this paper. After the information has been extracted from the UML system model, the second step ends at the connector `B` in Figure 1.

(3) Instantiating the Property
After the previous two steps have reached connectors `A` and `B`, the information extracted by the *UML Model Interpreter* is then used by the *Property Instantiator* to instantiate the structured natural language template with boolean propositions containing model-specific elements. In addition to specifying boolean expressions on variable values of UML classes, two other predicates are supported in the boolean propositions: (1) A `call(...)` predicate to specify that an signal of a class is called and (2) an `enter(...)` predicate to specify that a class enters a specific state.

In order to instantiate the property template for our running example, we need to replace P and S with appropriate boolean propositions generated from the information extracted from the system model in the previous step. The cause P

needs to describe that "*process()* of the Processor has been called" and is therefore replaced by `call(Processor.process())`. The effect S needs to capture that "the Processor returns to the Idle state", and is therefore replaced by `enter(Processor:Idle)`. Finally, we obtain the following instantiated natural language property:

> Globally, it is always the case that if `call(Processor.process())` holds, then `enter(Processor:Idle)` eventually holds.

(4) Analyzing the Property

After the instantiation step is completed, the model can be analyzed for adherence to the specified property. In SPIDER, the *UML Model Analyzer*, which is also part of the *Formal Analysis Tool Connector*, maps the instantiated natural language template to the corresponding specification pattern instances, namely LTL formulae [18] for the Spin model checker [16]. In order to enable the analysis of the property in our running example, the above instantiated natural language property is mapped to the following LTL formula:

$$\Box(\texttt{call(Processor.process())} \rightarrow \Diamond(\texttt{enter(Processor:Idle)}))$$

In order to reduce the cost of model checking, we perform an automated abstraction of the formal model before executing the analysis. The *UML Model Analyzer* performs a property-based slicing on the formal system model, where it invokes the slicing algorithm provided by Spin[2] and removes constructs identified as redundant. After the slicing is complete, SPIDER invokes the Spin model checker and performs the analysis.

(5) Displaying Analysis Results

After the model checking has completed, the *UML Model Analyzer* provides analysis results back to the *Property Instantiator*, which are then visually presented to the user using a traffic light icon. Red indicates that the property was violated and a counter example is returned; Green indicates that the property holds for the selected model; and Yellow 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. If a violation of a property is found, then the user can visualize the execution that lead to the violation, correct the model, and repeat the analysis. Finally, when the property holds on the selected model, the user can analyze additional properties or exit from the tool and the analysis process.

3 Related Work

Several tools have been developed for the formal analysis of system models specified in terms of UML, such as vUML [8], HUGO[/RT] [23], and Fujaba [10]. In addition,

[2] The slicing algorithm of Spin is sound and complete with respect to any property specifiable in terms of LTL [16].

some commercial tools commonly offer validation and/or animation capabilities, such as Rhapsody [6] and Rational XDE [7]. While these tools also have similar purposes when compared to SPIDER configured with Hydra, they do not offer support for the specification of properties in terms of natural language. On the other hand, numerous approaches [5] 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 [24]), 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. In summary, none of these approaches combines the completeness of a pattern 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-circular grammar can be constructed.

Several other projects have investigated how to make specification patterns more accessible via more informal representations. Smith *et al.* developed Propel [25], where they extended the specification patterns by Dwyer *et al.* [11] 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, and natural language templates help a specifier to precisely capture a property in natural language. In contrast to our approach, they focus on capturing subtle properties of individual specification patterns, rather than applying the specification patterns to the analysis of UML models. 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.

4 Conclusions

We have presented a configurable process for UML model analysis implemented in the SPIDER toolkit. We expect several benefits to be gained from using SPIDER. First, users less experienced in the specification of formal properties are able to create formally-analyzable natural language representations of properties for their UML models. Feedback from industrial collaborators has indicated that this specification style is preferred over formal specification languages. Second, SPIDER is extensible to the use of several formal analysis tools by offering the ability to plug in additional *Formal Analysis Tool Connector* components. Therefore, a wide variety of formal analysis tools can be used to analyze the behavioral properties. Currently, SPIDER supports the Spin model checker [16] for UML models and support for additional formal analysis tools is being developed.

Third, SPIDER provides a single environment for specification construction and analysis. The tool suite enables a user to automatically analyze a system model and visualize the analysis results. Currently, our tool is targeted at the novice specifier, as evidenced by the step-by-step guidance during the derivation process and making the formal specification language transparent to the user. 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 [27] and we are investigating techniques to mitigate these problems, such as the use of multiple views and different levels of assistance for the derivation and instantiation tasks.

Future work will investigate how to incorporate previously developed real-time extensions to our formalization framework [28] and specification patterns [17]. This work will also examine how to best visualize the analysis results. Finally, we are continuing to work with industrial collaborators to obtain feedback on the usability of SPIDER.

References

1. Lutz, R.R.: Targeting safety-related errors during software requirements analysis. In: SIGSOFT'93 Symposium on the Foundations of Software Engineering. (1993)
2. Object Management Group: Model driven architecture. `http://www.omg.org/mda/` (2005)
3. Berenbach, B.: The evaluation of large, complex UML analysis and design models. In: Proceedings of the 26th International Conference on Software Engineering (ICSE'04), IEEE Computer Society (2004) 232–241
4. Cheng, B.H.C., Stephenson, R., Berenbach, B.: Lessons learned from metrics-based automated analysis of industrial UML models (an experience report). In: Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica (2005) 324–338
5. Konrad, S., Cheng, B.H.C.: Facilitating the construction of specification pattern-based properties. In: Proceedings of the IEEE International Requirements Engineering Conference (RE05), Paris, France (2005)
6. I-logix: Rhapsody (2005) `http://www.ilogix.com/rhapsody/rhapsody.cfm`
7. IBM: Rational Rose XDE Developer (2005) `http://www-306.ibm.com/software/awdtools/developer/rosexde/`.
8. Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models. In: Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE99), Washington, DC, USA, IEEE Computer Society (1999)
9. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science **55**(3) (2001)
10. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proceedings of the 22nd International Conference on Software Engineering, New York, NY, USA, ACM Press (2000) 742–745
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society Press (1999) 411–420
12. McUmber, W.E., Cheng, B.H.C.: A general framework for formalizing UML with formal languages. In: Proceedings of the IEEE International Conference on Software Engineering (ICSE01), Toronto, Canada (2001)

13. Object Management Group: UML Specifications, Version 1.4 (2002) http://www.omg.org/cgi-bin/doc?formal/04-07-02.
14. Object Management Group: UML 2.0 Superstructure Specification (2004) http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.
15. Object Management Group: OMG-XML metadata interchange (XMI) specification, v1.1 (2000) http://www.omg.org/cgi-bin/doc?formal/00-11-02.
16. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts (2004)
17. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the International Conference on Software Engineering (ICSE05), St Louis, MO, USA (2005)
18. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc. (1992)
19. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (2) (1986) 244–263
20. Ramakrishna, Y.S., Melliar-Smith, P.M., Moser, L.E., Dillon, L.K., Kutty, G.: Interval logics and their decision procedures: Part I + II. Theoretical Computer Science **166;170**(1-2) (1996) 1–47;1–46
21. Olender, K.M., Osterweil, L.J.: Cecil: A sequencing constraint language for automatic static analysis generation. IEEE Transactions on Software Engineering **16**(3) (1990) 268–280
22. Campbell, L.A., Cheng, B.H.C., McUmber, W.E., Stirewalt, R.E.K.: Automatically detecting and visualizing errors in UML diagrams. Requirements Engineering Journal **7**(4) (2002) 246–287
23. Knapp, A., Merz, S., Rauh, C.: Model checking timed UML state machines and collaborations. In Damm, W., Olderog, E.R., eds.: 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002). Volume 2469 of Lecture Notes in Computer Science., Oldenburg, Germany, Springer-Verlag (2002) 395–414
24. R. Nelken, N. Francez: Automatic translation of natural-language system specifications into temporal logic. In Rajeev Alur, Thomas A. Henzinger, eds.: Proceedings of the Eighth International Conference on Computer Aided Verification CAV. Volume 1102., New Brunswick, NJ, USA, Springer Verlag (1996) 360–371
25. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: Propel: An approach supporting property elucidation. In: Proceedings of the 24th International Conference on Software Engineering, ACM Press (2002) 11–21
26. Mondragon, O., Gates, A.Q.: Supporting elicitation and specification of software properties through patterns and composite propositions. International Journal on Software Engineering and Knowledge Engineering **14**(1) (2004) 21–41
27. Khwaja, A.A., Urban, J.E.: Syntax-directed editing environments: Issues and features. In: SAC '93: Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing, ACM Press (1993) 230–237
28. Konrad, S., Campbell, L.A., Cheng, B.H.C.: Automated analysis of timing information in UML diagrams. In: Proceedings of the Nineteenth IEEE International Conference on Automated Software Engineering (ASE04), Linz, Austria (2004) 350–353 (Poster summary).