

Visualizing Requirements in UML Models*

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

Abstract

As the Unified Modeling Language (UML) and model-driven development (MDD) become increasingly common in industry, many developers are faced with the difficult task of understanding how an existing UML model realizes system requirements. Essentially, developers are required to understand the structure and behavior of UML models that they may have not created. Understanding these relationships is non-trivial, because the interactions in the model are not readily apparent. Commonly, the only means to elicit these relationships is visual inspection and guided simulation. This paper describes an alternative approach termed REVU (Requirements Visualization of UML), a process for visualizing functional requirements in terms of behavioral interactions in a UML model. We illustrate the use of this process with the visualization of scenarios for an adaptive light control system.

1. Introduction

As the Unified Modeling Language (UML) [29] and model-driven development (MDD), such as model-driven architecture (MDA) [29], become increasingly common in industry, many developers are faced with the difficult task of understanding how an existing UML model realizes system requirements. This understanding is important for developing new systems or maintaining and modifying legacy systems. Essentially, developers are required to understand the structure and behavior of UML models that they may

have not created. Understanding the relationship between a model and its requirements is non-trivial, because the interactions in the model are not readily apparent. This paper describes REVU (Requirements Visualization of UML), a process for visualizing functional requirements in terms of behavioral interactions in a UML model.

Two related approaches exist for analyzing behavioral relationships in UML models: visual inspection and guided simulation. Visual inspection means that the determination of how a system implements a requirement is achieved by looking at the UML model and attempting to understand the possible interactions in the model. Guided simulation [2, 13, 16, 28, 31] enables a developer to visualize an execution path through a model, but requires the developer to select the values of model variables at each branch in the execution path. As such, the developer must be fairly familiar with the details of the system. Furthermore, the very information that the developer is attempting to ascertain must already be known.

In contrast, the REVU process enables a developer with minimal knowledge of a model to learn about the system. Specifically, a critical aspect for understanding how a UML model realizes requirements are *witness scenarios*. A witness scenario is a particular sequence of steps executed by the system to fulfill a given functional requirement. The REVU process uses three key steps to create and visualize witness scenarios. First, the developer declaratively specifies properties of a witness scenario in natural language. The declarative specification allows a developer to “under-specify” the witness scenarios, thereby uncovering scenarios when little system information is available. Second, a model checker generates one or more witness scenarios that adhere to the previously specified properties. For this step, we use a model checker’s ability to search for counter examples in the state space and store these counter examples in the form of violation traces. Specifically, we exploit this trace creation capability to create witness traces,

*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 a grant from Michigan State University’s Quality Fund.

[†]Please contact this author for all correspondences.

which our tool suite automatically converts to witness scenarios in terms of UML elements. Third, the developer views each witness scenario in terms of the original UML model. Therefore, our process has three main contributions: (1) support for declaratively specifying the scenarios to be visualized in natural language; (2) automated generation of witness scenarios; and (3) support for visualizing the witness scenarios.

The REVU process leverages a roundtrip-engineering approach to the construction of conceptual UML diagrams for modeling and analyzing system requirements. In this process, requirements are specified declaratively as natural language properties that are mapped to temporal logic representations. In contrast, the system model is specified in operational fashion using UML class and state diagrams. Three existing approaches and their tool support jointly facilitate the round-trip engineering approach. First, the SPIDER tool suite [19, 21] facilitates the specification of formally analyzable properties using natural language. Properties specified using SPIDER can be automatically translated to formal specification languages, *e.g.*, linear-time temporal logic (LTL) [25] to be used with the model checker Spin [14]. Second, Hydra [26] implements a metamodel-based approach to mapping UML diagrams to the specification languages of formal analysis tools, *e.g.*, model checkers. Third, Theseus [8] visually animates the analysis results (*i.e.*, witness scenarios) in terms of the original UML diagrams. For the purpose of this paper, we assume this roundtrip-engineering approach was used during the creation of the UML model to analyze the model for adherence to its functional requirements specifications. As such, the cost of applying the REVU process is minimal; the major effort of constructing a system model adhering to its requirements specifications has already been done.

To validate our work, we applied the REVU process to an industrial case study for an adaptive light control system. In this case study, we demonstrate how REVU facilitates the understanding of complex system interactions. The remainder of the paper is organized as follows. Section 2 provides background information on the supporting elements of the REVU process. Section 3 gives a description of the REVU process. Section 4 presents the adaptive light controller case study. Section 5 overviews related work. Finally, Section 6 gives concluding remarks and discusses future investigations.

2. Background

This section provides a brief introduction to our natural language specification tool suite (SPIDER), our UML formalization framework (Hydra), and our UML visualization tool (Theseus).

2.1 Spider

SPIDER [19, 21] (Specification Pattern Instantiation and Derivation EnviRonment) is a process with corresponding tool support that enables a developer to use a natural language-based grammar to specify properties of UML models. Specifically, these properties are specified in natural language using a previously developed process for deriving and instantiating formally analyzable natural language properties based on specification patterns [5, 20]. Briefly, the SPIDER process comprises three steps:

1. **Derivation:** Derive a natural language sentence from a structured natural language 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 this process is a structured natural language grammar. This grammar is used to derive natural language sentences that can be mapped to formal specifications, such as LTL formulae [25].

2.2 Hydra

It is well-known that UML lacks a precise, formally defined semantics. Therefore, numerous semantic interpretations are possible for a given diagram. In order to address this problem and to make UML diagrams amenable to rigorous analysis, McUmbler and Cheng [26] developed a metamodel-based formalization framework that maps a given UML model into a formal specification language, such as Promela, the specification language for the model checker Spin [14]. Hydra automates this mapping process [26].

The general UML-to-Promela formalization approach is to map objects to processes in Spin (*proctypes*) that exchange messages via *channels*. Nested and concurrent states are also formalized as processes. Additional details on the modeling and analysis process, and the underlying formalization framework can be found in [26].

2.3 Theseus

Theseus [8] is a UML visualization tool that enables developers to visualize analysis results produced by formal analysis tools in terms of UML model elements. Theseus comprises an analysis-tool-specific *Theseus trace processor* and the generic *Theseus visualization engine*. The *Theseus trace processor* converts a trace created by a formal analysis tool, *e.g.*, a counter example generated by Spin [14], into a scenario in terms of the original UML model elements.

The scenario information includes state transitions and detailed information about messages passed between classes. This scenario is stored in a well-defined intermediate XML representation. The *Theseus visualization engine* reads the scenario from the XML file and presents a step-by-step animation of the counter example through both sequence and state diagrams. The developer has the option of viewing the entire counter example or stepping through any section of interest. The *Theseus visualization engine* is general in that it can visualize a scenario generated by any formal analysis tool, provided the scenario is represented in the well-defined XML format.

3. ReVU Process

In the following, we provide an overview of the REvU process, which enables developers to visually familiarize themselves with existing UML models of system requirements. Figure 1 depicts the REvU process in terms of a UML activity diagram. Specifically, the process comprises the following steps: (1) specifying the witness scenario to be visualized in natural language, (2) creating a formal representation of the UML model and natural language specification, (3) generating witness traces, (4) translating these witness traces to witness scenarios, (5) and visualizing these scenarios in terms of the original UML model. This process is explained in detail in the remainder of this section. For the purposes of this paper, Hydra and SPIDER are configured to read UML 1.4 [29] models¹ specified in terms of XMI 1.1 [29] and generate Promela specifications to be used by the model checker Spin [14].

3.1. Specifying Witness Scenario Properties

In REvU, the developer initially specifies a property describing the witness scenario to be visualized. In order to facilitate the specification process, we developed a natural language grammar for the specification of properties of witness scenarios for requirements. This grammar (shown in Figure 2) can be used by SPIDER to guide the developer in deriving a natural language property in a syntax-guided fashion [19].

While the grammar follows the structure of the specification pattern system by Dwyer *et al.* [5], LTL property templates that are not well-suited for the specification of witness scenarios have been modified. For instance, the LTL representation of the *response* specification pattern by Dwyer *et al.* is

$$\Box(P \rightarrow \Diamond S), \quad (1)$$

¹We do not make use of action semantics, the main distinction between UML 1.5 [29] and 1.4. For the recently finalized UML 2.0 [29], CASE tool support is still limited.

read as “Always P implies eventually S”. Clearly, this LTL formula is useful for finding *violation scenarios* of a response requirement, since every trace that does *not match the LTL property* represents a violation scenario of the requirement. However, this LTL formula is not suitable for uncovering *witness scenarios* of the same requirement, since any trace that *matches the LTL property* is not necessarily well-suited to serve as a witness scenario. Specifically, if P does not occur, Expression 1 is true. Thus, the model checker may uncover a trace without an occurrence of P. Such a trace would not be helpful to a developer attempting to understand how the system responds to an occurrence of P with an occurrence of S.

To make the LTL formulae useful for uncovering witness scenarios, we modified the specification patterns by Dwyer *et al.* that were poorly suited for the specification of witness scenarios. For example, we use the following LTL formula for uncovering a witness trace of a response requirement:

$$\Diamond(P \wedge \Diamond S), \quad (2)$$

read as “Eventually P and eventually S”. Using this modified LTL formula, only witness traces in which P occurs and this occurrence is eventually followed by an occurrence of S are uncovered. Where necessary, the remaining specification patterns instances were similarly modified.

The grammar contains natural language representations of all stutter-invariant specification patterns by Dwyer *et al.* [5],² tailored to the specification of witness scenario properties. According to the survey in [5], this selection of specification patterns is sufficient to specify more than 90% of properties encountered in practice. In the grammar, literal terminals are delimited by quotation marks (“ ”), non-literal terminals are given in a *sans serif* font, and non-terminals are given in *italics*.

Using this grammar, developers are able to derive natural language templates of properties that can be mapped to the temporal logic representations of the corresponding specification pattern. For example, the natural language representation of a *response property* states:

Globally, it is eventually the case that P is eventually followed by S. (Grammar: 1, 2, 3, 8, 10) (3)

Before instantiating a natural language property to be applicable for the system model at hand, the developer replaces the placeholders (*i.e.*, P and S in Expression 3) with free-form text describing the system conditions to which the property applies. In our running example, assume that the placeholder P is replaced with a malfunction is detected and

²We exclude the *Precedence Chain*, *Response Chain*, and *Constrained Chain* patterns of Dwyer’s specification patterns as these patterns are not closed under stuttering [4], an important property for efficient analysis with Spin [14].

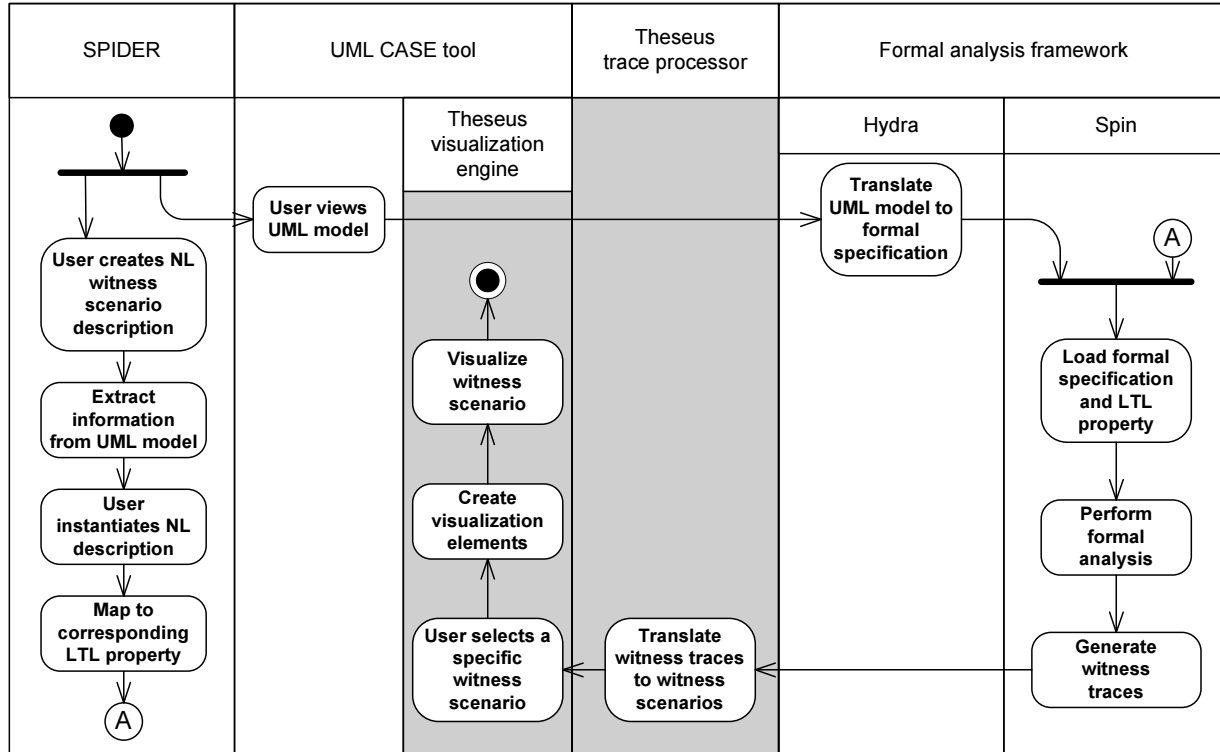


Figure 1. Activity diagram depicting the ReVU process

Start	1: property	::= <i>scope</i> “,” <i>specification</i> “.”
Scope	2: scope	::= “Globally” “Before ” R “After ” Q “Between ” Q “ and ” R “After ” Q “ until ” R
General	3: specification	::= “it is ” (<i>occurrenceCategory</i> <i>orderCategory</i>)
	4: occurrenceCategory	::= (<i>absencePattern</i> <i>universalityPattern</i> <i>existencePattern</i> <i>boundedExistencePattern</i>) “ the case that ” P
	5: absencePattern	::= “never”
	6: universalityPattern	::= “always”
	7: existencePattern	::= “eventually”
	8: orderCategory	::= “eventually the case that ” P (<i>precedencePattern</i> <i>responsePattern</i>)
	9: precedencePattern	::= “, when previously ” S
	10: responsePattern	::= “ is eventually followed by ” S

Figure 2. Structured English grammar

the placeholder S is substituted for the free-form text the malfunction indicator light being turned on to obtain the following natural language property:

Globally, it is eventually the case that a malfunction is detected is eventually followed by the malfunction indicator light being turned on. (4)

In the next step, the property is instantiated for the given UML model by replacing the free-form text with boolean propositions on system conditions. The UML model is developed with existing tools, such as ArgoUML [32] or Rational Rose XDE [17], and exported to XMI [29]. To fa-

cilitate the instantiation process, SPIDER extracts model information from the XMI representation of the UML diagrams. This information may be selected during the property instantiation process to automatically generate templates for system conditions that can then be inserted into the placeholders to replace the free-form text. Specifically, these templates facilitate the specification of conditions on variable values, state machine locations, and method call events. For our running example, assume the placeholder a malfunction is detected is instantiated as Detector.malfunction=1 and the malfunction indicator light being turned on is replaced with Mil.status=1,

Globally, it is eventually the case that `Detector.malfunction=1` is eventually followed by `Mil.status=1`. (5)

which is automatically mapped by SPIDER to the following LTL property

$$\diamond((\text{Detector.malfunction}=1) \wedge \diamond(\text{Mil.status}=1)). \quad (6)$$

At the end of this step, the developer has created a formally analyzable specification of the scenario to be visualized in the UML model.

3.2. Generating Witness Scenarios

Next, the LTL property created with SPIDER is used by a formal analysis tool, *i.e.*, Spin [14], to create a witness trace. Model checkers, in general, search for violations of a property by traversing the complete state space of the system, thereby analyzing every possible execution sequence. If a sequence is uncovered that violates the specified property, then the model checker provides this sequence as a violation trace, which may be used by our visualization framework and Theseus for the visualization of a property violation (this process is described in [8]).

In contrast to the previous technique, this work presents the generation and visualization of witness scenarios in terms of the original UML model. The witness scenarios are created from witness traces generated by the model checker. A witness trace, as opposed to a violation trace, represents a sequence of steps executed by the system that adheres to the specified property, instead of violating it. In this paper, we describe how the model checker Spin [14] can be used to create such witness traces for a property. Specifically, we specify the complement of the LTL property. For instance, the complement of Expression 2 is

$$\neg(\diamond(P \wedge \diamond S)) \equiv \square(\neg(P \wedge \diamond S)), \quad (7)$$

read as “Globally, it is never the case that P followed by S”. This complementary property is passed to the model checker. If the model checker finds a violation trace of the complementary property, then the uncovered trace is a witness trace of the original property. Since model checkers are most effective in finding and generating violation traces [7], the model checker commonly finds such traces traversing only a small portion of the model’s state space. However, if the model checker does not detect a witness trace for the property in the state space, then no trace in the system model adheres to the functional requirement of interest. In general, developers may need to revise such a system model. For example, the environment model of the system may need to be extended so that the system exhibits the desired witness scenarios. The modification of the

model should be done using our round-trip UML modeling approach, so that the model adheres to its critical properties after the modification is completed.

A single witness scenario may not be sufficient to completely understand how the system model captures a requirement, since it only shows a single instance of such a system execution. Rather, it may be necessary to visualize several witness scenarios to understand how a requirement is modeled in a system. Therefore, our process offers three modes for generating different witness traces, from which different witness scenarios may be created.

- 1) **First traces:** In this mode, the first x witness traces encountered by the model checker are stored. While this mode is the fastest, the generated witness traces are often unnecessarily long and complex, since they represent the first violations encountered in a depth-first search. (This assessment corresponds with the findings presented in [7].)
- 2) **Shortest trace:** Spin offers an algorithm for iteratively finding the shortest trace [14]. The shortest witness trace for a property is the minimal behavior the system performs for a requirement. It excludes irrelevant steps that may be performed by concurrent components. Therefore, shorter traces are commonly easier to understand. It is important to note that Spin only guarantees this trace to be the shortest for safety properties [14]. However, in our experience, the algorithm used by Spin is generally successful in finding a witness trace that is considerably shorter than those created by the *first traces* mode for safety as well as liveness properties. Although, for a liveness property, this trace may not be the shortest possible.
- 3) **Bounded search:** In this mode, the depth of the state space tree of the system that the model checker traverses is limited and all witness traces within this part of the state space are stored. This mode offers a more complete view of the system behavior for a requirement, since all witness traces starting from the root of the tree to a limited depth are stored. However, in this mode, the selection of an appropriate concrete search depth is crucial for the search effectiveness. If the search depth is too shallow, then only a small number of witness traces or no witness traces may be uncovered. Whereas, if the search depth is too large, then the number of witness traces may be quite numerous. In our experience, it is best to address this issue by starting with a shallow search depth and incrementally deepening it.

After the witness traces are generated by the model checker, the Theseus trace processor converts the traces into scenarios in terms of the original UML model. Subsequently, the Theseus visualization engine parses the scenarios to extract information that can aid a developer in selecting a scenario to visualize. Specifically, it displays

the number of objects in a scenario and the total count, as well as the individual count for every object, of how many transitions, send, and receive operations are performed (for more details, please refer to [23]). With this information, a developer can gather high-level information about a scenario before performing the actual visualization. For example, based on this information, the developer may choose to view the shortest scenario, or a scenario that includes interaction with a specific object of interest.

3.3. Visualizing witness scenarios

Theseus [8] offers two different animation modes for a scenario: *state diagram animation* (shown in Figure 3) and *sequence diagram generation* (for a screen capture showing a generated sequence diagram please refer to [23]). The state diagram helps in understanding the internal transitions performed by a single object during a collaboration. Additionally, the sequence diagram generated by Theseus is suited for illustrating the collaboration between several objects when performing an execution sequence for a specific requirement.

The developer is given three options when visualizing a scenario: skip to a specific step in the scenario, iterate through the scenario one step at a time, or automatically run through all the steps. For each step, Theseus animates the UML state diagrams and generated sequence diagram. Specifically, for the state diagrams, Theseus colors the current state and fired transitions red, and previously visited states yellow. This animation helps a developer to visually ascertain what steps the model has taken to achieve the behavior being examined.

4. Example Application of ReVU

In the following, we apply the ReVU process to an adaptive light control system.³ For brevity, we assume the developer has already determined the necessary elements for specifying the scenario, *i.e.*, the developer does not need to go through the preliminary steps of “underspecifying” the scenario and refining the specification. In addition, we assume that the *shortest trace* mode is always used for generating witness traces. We visualize a requirement that illustrates how ReVU greatly facilitates the understanding of complex system interactions that may be difficult to understand by other means, such as guided simulation or visual inspection.

³Due to space constraints, we only show the visualization of one requirement. For more details, please refer to [23].

4.1. Adaptive Light Control System

The adaptive light control system (ALCS) is responsible for moderating the light in a room. A class diagram depicting the structure of this system is depicted in Figure 4, where class attributes and operations have been elided for brevity. This UML model for ALCS was constructed using object analysis patterns.⁴ The primary function of the ALCS is to ensure that if a room is occupied, then the room is sufficiently illuminated, either by natural light or by lamps connected to a dimmer. The ALCS comprises a switch for manually turning on the lamps, a display for communicating messages to a user (the switch and display are both combined in the `UserInterface`), a `MotionSensor` for detecting whether the room is occupied, a `BrightnessSensor` for detecting the current illumination level of the room, a `Dimmer` that controls the brightness of the lamps, and a `ComputingComponent` to compute the required dimmer value. In the system model, the `Environment` is used to simulate different brightness levels (measured by the `BrightnessSensor`) as well as the user entering and exiting the room (detected by the `MotionSensor`). In addition, the `Dimmer` affects the brightness value in the `Environment` that is in turned reported to the `BrightnessSensor`.

4.2. Sample Requirement Visualization

A fundamental requirement for the system states that “If the room is occupied and the system is in automatic mode, then the system shall moderate the brightness in the room to achieve the user-selected brightness level.”

Specifying Witness Scenario Properties: Assume the developer is interested in visualizing the following witness scenario for the aforementioned requirement: “A user enters the room, while the system is in automatic mode and the brightness level in the room is less than the desired brightness. In response, the system activates the dimmer to achieve the desired brightness in the room.”

To visualize this requirement, the developer first derives the natural language template for a response property from the grammar in Figure 2 using SPIDER:

Globally, it is eventually the case that \underline{P} is eventually followed by \underline{S} . (8)

In order to capture the intent of the property, the developer replaces the placeholder \underline{P} with the user entering the room while the system is in automatic mode and the brightness level is not sufficient and the placeholder \underline{S} with

⁴*Object analysis patterns* guide developers in the creation of conceptual models during the analysis phase that precedes the design phase. For more information on object analysis patterns, please refer to [22].

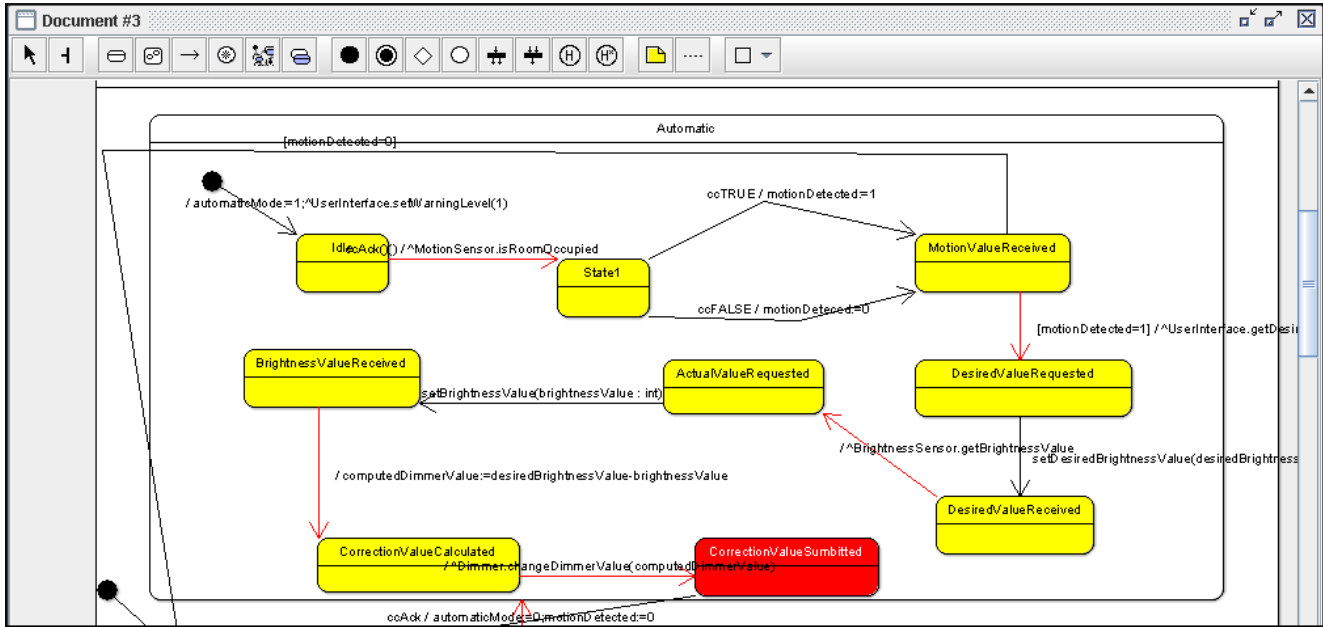


Figure 3. Screen shot of a state diagram animated by Theseus

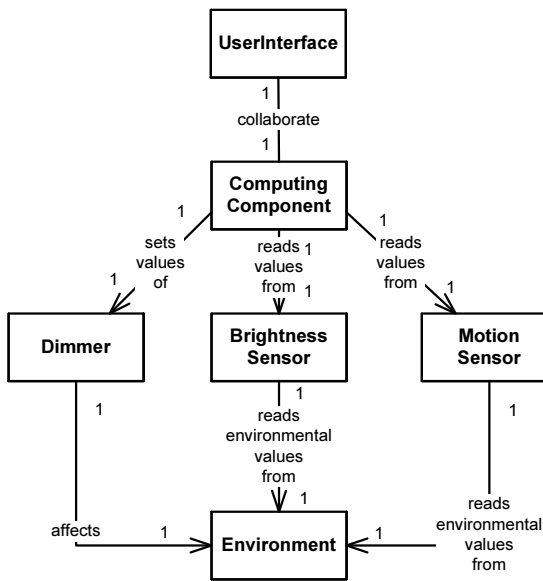


Figure 4. Class Diagram of the ALCS

the system regulating the dimmer to match the desired brightness level. As a result, the following natural language property is obtained:

Globally, it is eventually the case that the user entering the room while the system is in automatic mode and the brightness level is not sufficient is eventually followed by the system regulating the dimmer to match the desired brightness level. (9)

SPIDER facilitates the instantiation of the free-form description by providing a list of UML elements in the model and the capability to automatically generate templates for boolean propositions to replace the free-form text. By examining the UML model using the UML CASE tool and previous requirements visualizations, the developer has determined that when the system detects that the user enters the room and is in automatic mode, then the variables `ComputingComponent.motionDetected` and `ComputingComponent.automaticMode` are both set to *true*. To capture this condition, the boolean proposition `ComputingComponent.motionDetected=1 ∧ ComputingComponent.automaticMode=1` is created using SPIDER. In addition, the developer has determined that the desired brightness level is denoted by the integer variable `UserInterface.desiredBrightnessValue` and the actual brightness level in the room is denoted by the integer variable `Environment.brightnessValue`. Therefore, an insufficient brightness level is expressed by the condition `Environment.brightnessValue < UserInterface.desiredBrightnessValue` and a successful regulation will be represented by the condition `Environment.brightnessValue = UserInterface.desiredBrightnessValue`. Thus, the free-form text the user entering the room while the system is in automatic mode and the brightness level is not sufficient is replaced with `ComputingComponent.motionDetected=1 ∧ ComputingComponent.automaticMode=1 ∧ (Environment.brightnessValue < UserInterface.desiredBrightnessValue)` and

the free-form text the system regulating the dimmer to match the desired brightness level is replaced with `Environment.brightnessValue = UserInterface.desiredBrightnessValue`. As a result, the following instantiated natural language property is obtained:

Globally, it is eventually the case that `ComputingComponent.motionDetected=1` \wedge `ComputingComponent.automaticMode=1` \wedge (`Environment.brightnessValue < UserInterface.desiredBrightnessValue`) is eventually followed by `Environment.brightnessValue = UserInterface.desiredBrightnessValue`. (10)

Generating Witness Scenarios: SPIDER automatically translates the property shown in Expression 10 to LTL and invokes Spin, which quickly discovers the shortest witness trace for the property. Subsequently, Theseus processes the witness trace created by Spin and generates a witness scenario.

Visualizing Witness Scenarios: Theseus animates the UML model to depict the witness scenario. The sequence diagram visualization of this witness trace can be seen in Figure 5. Specifically, the sequence diagram visualizes the system behavior in a scenario described in Expression 9. First, the `ComputingComponent` checks whether the user has activated the manual mode in the `UserInterface`. Since the manual mode is not found to be active, the `ComputingComponent` sets the display mode of the `UserInterface` to *automatic mode* and continues by checking whether the room is occupied. The `MotionSensor` checks the `Environment` for user presence and sends the signal `ccTRUE()` to the `ComputingComponent`, thereby indicating that the room is occupied. The `ComputingComponent` requests the desired brightness value from the `UserInterface` and uses the `BrightnessSensor` to read the actual brightness value in the room from the `Environment`. Based on the actual brightness level and the desired brightness level, the `ComputingComponent` computes the required correction value and sends it to the `Dimmer`, which applies the correction to the `Environment`.

The visual depiction of this scenario expands the developer’s understanding of the requirement by highlighting the objects and interactions that can occur to fulfill it. For example, the developer now knows that the `ALCS` checks the `MotionSensor` in the room to determine whether the user is present, before querying the `UserInterface` and `BrightnessSensor` to determine the required correction value that will be applied to the `Dimmer`.

5. Related Work

Three primary categories of related work exist: (1) test suite generation, which is related to our approach since it also creates witness traces; (2) query checking, which is also intended to explore a formal system model; and (3) approaches to visual simulation and visualization of analysis results in UML models.

Several test generation approaches [1, 6, 7, 10, 12, 15, 30] use a similar trace generation process as that used by REVU. Specifically, these approaches use model checkers to generate witness traces for properties. These witness traces are then used to create test suites for testing the implementation of the system for adherence to its specification. Differing from our approach, their objective is not to visualize requirements in terms of UML model elements, but to generate test cases that adhere to coverage and optimality criteria.

Query checking [3, 9] typically specifies a temporal logic query with placeholders and then, using a model checker, discovers a solution to the query. A solution replaces the query placeholders with propositional formulae that are satisfied by the model. Query checking is a powerful approach to reverse engineering, since it enables a developer to uncover unknown properties of the model. However, in general, it is not concerned with the visualization of such properties. Therefore, query checking is a complementary approach, since REVU could be used to visualize properties that have been uncovered through query checking.

Numerous CASE tools [2, 13, 16, 28, 31] provide visual simulation support for systems specified in terms of UML models. Simulation, in contrast to REVU, requires the developer to have extensive knowledge about a system model in order to guide the simulation to achieve the visualization of a specific requirement. Other tools exist that visualize analysis results from model checkers in terms of UML models, such as `vUML` [24], `MOCES` [27], and `Hugo/RT` [18]. However, to the best of our knowledge, none of these tools is concerned with the generation and visualization of witness traces for the original UML diagrams.

A related approach that also uses a model checker to compute a feasible sequence of execution rather than a counter example is the `Play-In/Play-Out` methodology [11]. In this approach, a developer plays in scenarios of a system and a supporting tool creates live sequence charts (LSCs) specifying the behavioral requirements of the system. A smart play-out engine then can compute a single system response that does not violate universal charts, or a sequence that satisfies an existential chart. In contrast to our approach, the `Play-In/Play-Out` methodology is tailored to the specification of scenario-based system requirements, while our approach focuses on the automatic visualization of scenarios in UML class and state models. In addition, the `Play-`

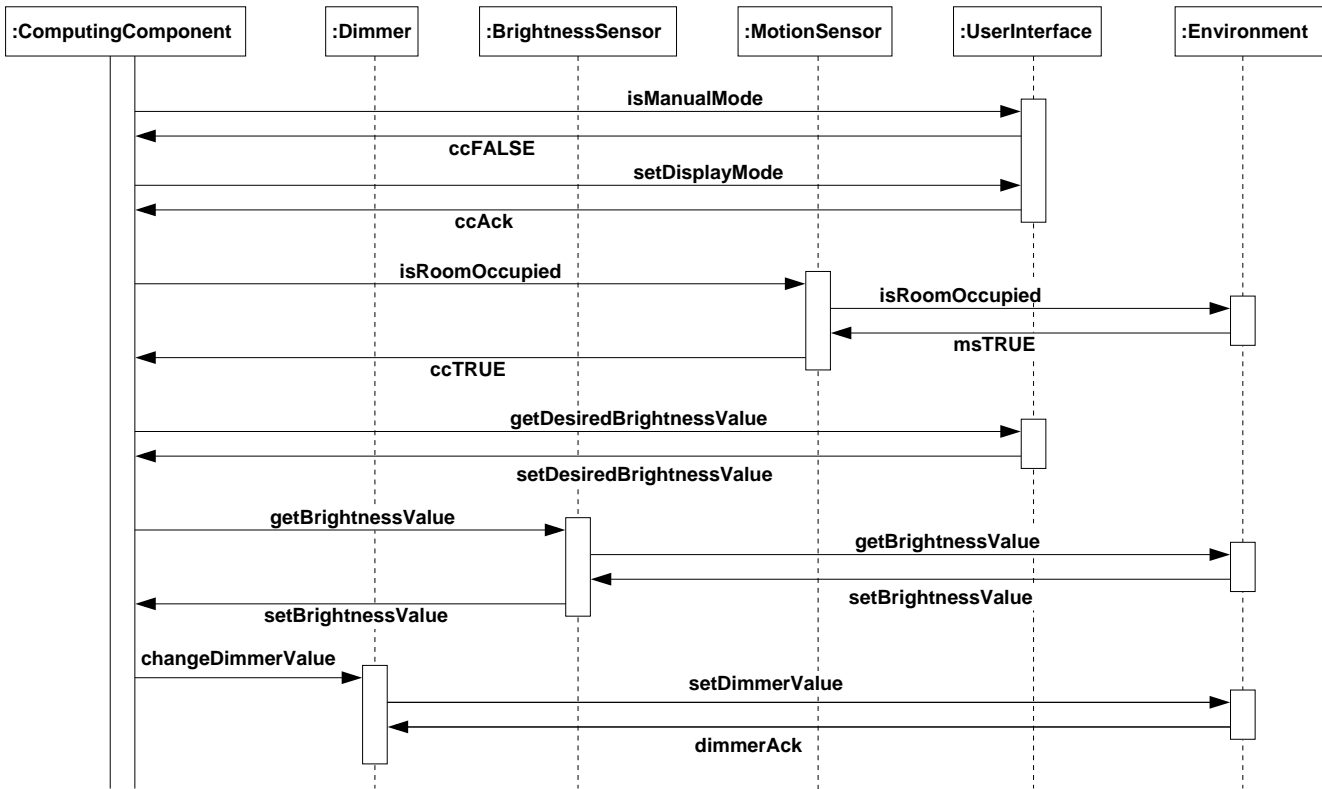


Figure 5. Sequence diagram visualizing an ALCS requirement

In/Play-Out methodology uses the same specification language, LSCs, to specify the system’s behavior and the requirement of interest, while our approach uses operational and declarative approaches.

In conclusion, none of the aforementioned approaches combines natural language specification of witness scenario properties, the automated generation of such witness scenarios, and the visualization of the witness scenarios in terms of the original UML model.

6. Conclusions

This paper described REVU, a requirements visualization process for UML models. REVU facilitates the visualization of witness scenarios in UML models by automatically generating such scenarios from natural language descriptions. Specifically, the capacity to describe such a scenario with as little information as is available makes our process well-suited for developers that need to understand a UML model, but did not initially create this model.

Numerous directions for future work are possible. First, the tool support could be extended to accommodate traces generated by formal analysis tools other than Spin. The traces could also be preprocessed to determine common-

alities, such as common prefixes and suffixes, which could facilitate the understanding of a scenario. Finally, the efficiency of REVU could be increased through the use of property-based slicing that automatically removes information not necessary for visualizing a witness scenario from the model. Therefore, this information would also be removed from the witness traces, potentially facilitating their understanding.

Acknowledgements

The authors thank the current and former faculty and students from the Software Engineering and Network Systems (SENS) Laboratory at Michigan State University, in particular Dr. Laura A. Campbell, Dr. William E. McUmbert, Min Deng, and Stephane Kamdoum.

References

- [1] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. of the Second IEEE Internat. Conf. on Formal Eng. Methods*, page 46, December 09-11 1998.
- [2] ARTISAN Soft.. Real-time Studio. <http://www.artisansw.com>, 2005.
- [3] W. Chan. Temporal-logic queries. In *CAV '00: Proc. of the*

- 12th Internat. Conf. on Computer Aided Verification, pages 450–463, London, UK, 2000. Springer-Verlag.
- [4] M. Chechik and D. O. Paun. Events in property patterns. In *Proc. of the 5th and 6th Internat. SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 154–167. Springer-Verlag, 1999.
- [5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of the 21st Internat. Conf. on Soft. Eng.*, pages 411–420. IEEE Computer Society Press, 1999.
- [6] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Proc. of the Third Internat. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes in Computer Science, pages 384–398. Springer, 1997.
- [7] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE-7: Proc. of the 7th European Soft. Eng. Conf. held jointly with the 7th ACM SIGSOFT Internat. Symposium on Foundations of Soft. Eng.*, pages 146–162, London, UK, 1999. Springer-Verlag.
- [8] H. Goldsby, B. H. C. Cheng, S. Konrad, and S. Kamdoum. Enabling a roundtrip Eng. process for the modeling and analysis of embedded systems. In *Proc. of the ACM/IEEE 8th Internat. Conf. on Model Driven Eng. Languages and Systems*, Genova, Italy, October 2006. (Accepted to appear).
- [9] A. Gurfinkel, M. Chechik, and B. Devereux. Temporal logic query checking: A tool for model exploration. *IEEE Transactions on Soft. Eng.*, 29(10):898–914, 2003.
- [10] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd Internat. Conf. on Soft. Eng. and Formal Methods*, pages 261–270, Beijing, China, Sept. 2004. IEEE Computer Society.
- [11] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Proc. of the 4th Internat. Conf. On Formal Methods in Computer-Aided Design (FMCAD'02)*, Portland, OR, 2002.
- [12] M. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *Third Internat. Workshop on Formal Approaches to Soft. Testing (FATES)*, pages 42–59, Montreal, Canada, October 2003.
- [13] W.-M. Ho, J.-M. Jézéquel, A. L. Guennec, and F. Pennaneac. UMLAUT: An extendible UML transformation framework. In *Proc. of the Automated Soft. Eng. Conf.*, Florida, October 1999.
- [14] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2004.
- [15] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *TACAS '02: Proc. of the 8th Internat. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341, London, UK, 2002. Springer-Verlag.
- [16] I-logix. Rhapsody and Statemate. www.ilogix.com/fs_prod.htm.
- [17] IBM. Rational Rose XDE Developer. <http://www-306.ibm.com/Soft./awdtools/developer/rosexde/>, 2005.
- [18] A. Knapp, S. Merz, and C. Rauh. Model checking timed UML state machines and collaborations. In W. Damm and E.-R. Olderog, editors, *7th Internat. Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002)*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–414, Oldenburg, Germany, Sept. 2002. Springer-Verlag.
- [19] S. Konrad and B. H. C. Cheng. Facilitating the construction of specification pattern-based properties. In *Proc. of the IEEE Internat. Requirements Eng. Conf. (RE05)*, Paris, France, August 2005.
- [20] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proc. of the Internat. Conf. on Soft. Eng. (ICSE05)*, St Louis, MO, USA, May 2005.
- [21] S. Konrad and B. H. C. Cheng. *Automated Analysis of Natural Language Properties for UML Models*, volume 3844 of *Lecture Notes in Computer Science*, chapter 6, pages 48–57. Springer-Verlag GmbH, January 2006.
- [22] S. Konrad, B. H. C. Cheng, and L. A. Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Soft. Eng.*, 30(12):970–992, December 2004.
- [23] S. Konrad, H. Goldsby, K. Lopez, and B. H. Cheng. Visualizing requirements in UML models. Technical Report MSU-CSE-06-23, Computer Science and Eng., Michigan State University, East Lansing, Michigan, June 2006.
- [24] J. Lilius and I. P. Paltor. vUML: A tool for verifying UML models. In *Proc. of the 14th IEEE Internat. Conf. on Automated Soft. Eng.*, page 255, Washington, DC, USA, 1999. IEEE Computer Society.
- [25] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.
- [26] W. E. McUumber and B. H. C. Cheng. A general framework for formalizing UML with formal languages. In *Proc. of the IEEE Internat. Conf. on Soft. Eng. (ICSE01)*, Toronto, Canada, May 2001.
- [27] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in Promela/Spin. In *WIFT '98: Proc. of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 90, Washington, DC, USA, 1998. IEEE Computer Society.
- [28] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *Proc. of the 22nd Internat. Conf. on Soft. Eng.*, pages 742–745, New York, NY, USA, 2000. ACM Press.
- [29] Object Management Group. <http://www.omg.org>, 2006.
- [30] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Internat. Conf. and Workshop on the Eng. of Computer Based Systems*, 2001.
- [31] Telelogic. ObjectGEODE. <http://www.telelogic.com/>, 2005.
- [32] Tigris.org. ArgoUML: The project home. <http://argouml.tigris.org/>, 2005.