A Visualization Framework for the Modeling and Formal Analysis of High Assurance Systems[*]

Heather Goldsby, Betty H.C. Cheng[**], Sascha Konrad, and Stephane Kamdoum

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, Michigan 48824 USA
{hjg, chengb, konradsa, kamdoumm}@cse.msu.edu

Abstract. Increasingly, object-oriented technology, specifically the Unified Modeling Language (UML), is being used to develop critical embedded systems. Several efforts have attempted to translate UML models into formal specification languages, thus enabling the models to be analyzed by model checkers. Unfortunately, the complexity and volume of the analysis results often prevents developers from fully taking advantage of the analysis capabilities. This paper introduces a generic visualization framework, Theseus, that provides developers with a model-based, visual interpretation of the analysis results in terms of the original UML diagrams. Within this framework, a playback mechanism displays the execution path that has led to a model checking violation in terms of the original UML state diagram and a newly generated sequence diagram that depicts the problem scenario. A Theseus prototype supporting the Spin and SMV model checkers has been applied to the analysis of UML models for embedded systems from industry.

1 Introduction

Embedded systems have become increasingly pervasive, particularly occurring in high-assurance systems, such as automotive systems, medical devices, and telecommunication systems. Given the critical nature of these embedded systems applications, it is important to use rigorous development techniques. Increasingly, object-oriented technology is being used to develop embedded systems [1]. Furthermore, the Unified Modeling Language (UML) [2], the *de facto* standard

[*] This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, CDA-9700732, CCR-9901017, CNS-0551622, CCF-0541131, 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. Stephane Kamdoum completed a portion of this work while studying under an International Exchange Program between the University of Kaiserslautern and Michigan State University.

[**] Corresponding author.

for object-oriented modeling, is the primary modeling notation for the recent movement towards model-driven development (MDD), such as that used in the model-driven architecture (MDA) by the OMG [2]. Using MDD, the models are refined iteratively from requirements to design and eventually code is generated. One drawback with the UML has been the lack of model analysis tools. To date, most of the UML analysis has been limited to syntactic-based analysis or simulation. Recently, there have been several efforts to translate object-oriented diagrams (e.g., state and sequence diagrams) to formal specification languages [3, 4, 5, 6] to be analyzed for adherence to behavioral properties by model checkers, such as Spin [7] and SMV [8]. A challenge with this approach to analysis is how to understand and then use the error descriptions from the analysis output to revise the original UML diagrams. This paper describes a generic visualization framework, Theseus, that interprets the analysis output from model checkers in terms of the original UML diagrams. Using Theseus, the developer is alleviated from the burden of deciphering the frequently cryptic and verbose trace output, which is often denoted in an analysis tool-specific language, including references to line numbers of the specification, internal process numbers, temporary variable names, etc.

In addition to the syntactic-based analysis tools, such as those provided with XDE [9], several CASE tools [10, 11, 12, 13, 14] provide visualization support for (UML) model simulation. Simulation provides information about a single execution path (e.g., a scenario) through a system model, where visualizations can be used to depict a scenario by displaying message traces in sequence diagrams or highlighting elements of a state diagram. Simulation-based analysis *validates* that a model conforms to a developer's expectations. In contrast, the recent work of translating the UML diagrams to model checker specification languages is intended to support the *verification* of UML models. That is, does a UML model satisfy temporal properties, such as invariants and leads-to properties, for all possible execution paths. Particularly for high-assurance systems, it is important to be able to verify a UML model against critical properties *before* the models are refined to design and code. A notable feature of model checkers is that if a system model does violate a property, a counterexample depicting the sequence of events and/or states causing the violation is returned. Two challenges exist with using the analysis results. First, a developer must decipher the verbose and often non-intuitive representation of system elements specified in the counterexample. Second, the cause of the error must be traced back to the original UML diagram in order to make the appropriate model refinements, particularly in the context of MDD.

This paper describes a generic visualization framework, Theseus, that supports a model-driven, visual interpretation of analysis output from commonly used model checkers. Three tasks were essential in the development of Theseus. First, based on numerous trace output files generated from each model checker, we constructed a grammar and a corresponding parser for each model checker to be supported by Theseus; the parser generates an abstract syntax graph (ASG) for a given trace file. Second, we developed a translator for each formal analysis tool that traverses the ASG to generate a generic XML representation containing only UML-relevant model elements, such as state names, transition names, attributes, etc. The parser and translator are combined into an analysis tool-specific trace processor. Third, we developed a visualization engine that processes the XML representation of the counterexamples to support UML state diagram animation and sequence diagram generation. The combination of these three elements have been encompassed in the Theseus prototype that accepts as input a UML model and the trace file for a counterexample generated from a model checker for an error detected in the UML model, and produces a state diagram animation and sequence diagram depicting the counterexample. The user has the option of either stepping (single or multi-step) through the animation or running through the complete counterexample, where color changes are used to depict state and transition traversals.

Theseus has been developed to provide a critical piece of a larger project supporting a roundtrip-engineering approach to the construction of UML diagrams for modeling and analyzing embedded systems requirements. Specifically, we have previously developed several techniques and tools to provide a bridge between (semi-)informal and formal approaches to requirements engineering of embedded systems. First, in order to enable UML diagrams to be automatically analyzed by model checkers, we developed a meta-model based approach to mapping UML diagrams to target specification languages [3]. Hydra is a prototype tool that supports the automatic generation of specification languages, such as Promela, the specification language of the Spin model checker [7], from UML class and state diagrams. Second, in order to help developers create the UML diagrams, we developed a set of object analysis patterns for embedded systems [15], that provide sample structural and behavioral UML templates for modeling embedded systems. Third, in order to facilitate the specification of formally analyzable properties using natural language, we have developed a structured natural language grammar and SPIDER (**S**pecification **P**attern **I**nstantiation and **D**erivation **E**nvi**R**onment) [16, 17]. Using SPIDER, developers can create natural language specifications of properties that are automatically and transparently mapped to the property specification language of the targeted analysis tools, *e.g.*, linear-time temporal logic (LTL) [18] for the model checker Spin [7]. Theseus provides the fourth component of the roundtrip-engineering process, that is, the visualization of the model checking analysis. Therefore, putting all four elements together, a developer can use the object analysis patterns to create a UML model for an embedded system, use Hydra to generate a formally analyzable model for a model checker, use SPIDER to specify properties to be satisfied by the UML model, use the model checker to analyze the UML model against the SPIDER-specified properties, and use Theseus to visualize counterexamples generated from the model checker in terms of the original UML diagrams, thus completing the roundtrip-engineering process.

In order to validate our work, Theseus has been instantiated to handle trace output generated from two different model checkers, Spin and SMV, and we have applied our roundtrip-engineering process to the analysis of several industrial embedded systems. The remainder of the paper is organized as follows. Section 2 provides background information on the supporting elements of the roundtrip-engineering process. Section 3 gives the architecture for Theseus and describes the visualization capabilities. Section 4 presents a case study involving the Spin model checker results. Section 5 overviews related work. Finally, Section 6 gives concluding remarks and discusses future investigations.

2 Roundtrip Modeling and Analysis Overview

This section introduces the roundtrip modeling and analysis process depicted in **Fig. 1**, where the shaded swimlanes depict the activities encompassed by Theseus. Specifically, we describe the process of creating a UML model, formalizing the model, and checking the model for adherence to properties.

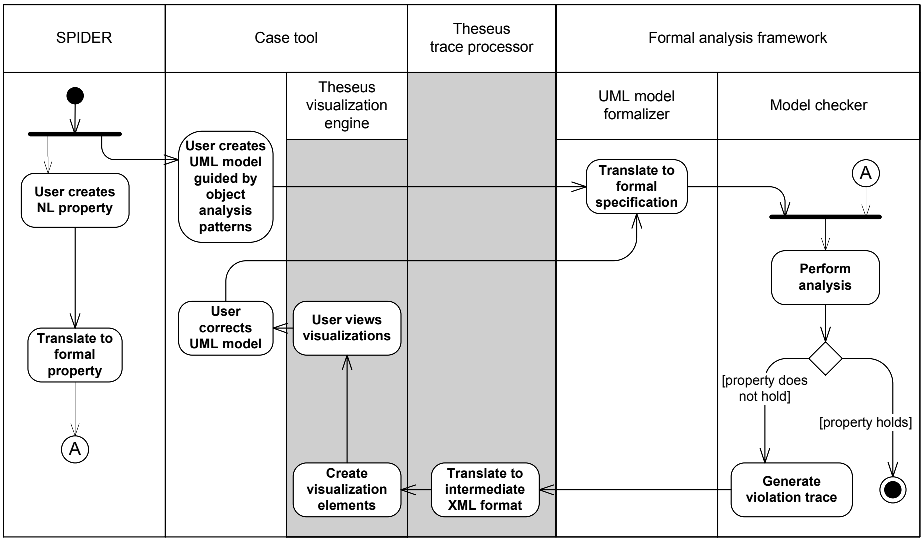

SPIDER
Case tool
Theseus trace processor
Formal analysis framework
Theseus visualization engine
UML model formalizer
Model checker
User creates UML model guided by object analysis patterns
User creates NL property
Translate to formal specification
A
Perform analysis
User corrects UML model
User views visualizations
Translate to formal property
[property does not hold]
[property holds]
A
Create visualization elements
Translate to intermediate XML format
Generate violation trace

Fig. 1. Roundtrip Modeling and Analysis Process

2.1 Step 1: Creating a UML Model and Specifying a Property

In the first step, the developer uses a CASE tool, such as ArgoUML [19], to create a UML model that describes the structure and behavior of the system. In general, the structure of the system is described in terms of UML class diagrams. Behavioral aspects are modeled using state diagrams associated with the classes. Abstraction should be used to address the size and complexity of the model. Specifically, we model only those portions of the system that are relevant to the analysis. Multiple, specialized models can be created for different aspects of the system.

To aid in the creation of these models for embedded systems, we previously developed *object analysis patterns* [15]. Whereas design patterns [20] guide developers in the construction of design models, object analysis patterns guide developers in the creation of conceptual models during the analysis phase preceding the design phase. Specifically, these patterns aid in the construction of conceptual models of the embedded systems focusing on functional aspects, where these models may later be refined in the design phase through the use of design patterns.

Next, the user specifies the properties of the UML model to be analyzed. In our approach, these properties are specified in natural language using a previously developed process for deriving and instantiating formally analyzable natural language properties based on real-time and qualitative specification patterns [16, 17], termed Spider. 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 structured in terms of a specification pattern system. In this paper, we use the qualitative portion of a previously developed structured English grammar [21] for the specification patterns by Dwyer *et al.* [22].

Using Spider, the developer specifies the property to be verified in natural language. Spider then translates the natural language property to a form that can be understood by the targeted analysis tool.

2.2 Step 2: Formalizing a UML Model

The UML model created from the object analysis patterns is translated into the specification language for the targeted model checker. 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, McUmber and Cheng [3] developed a metamodel-based formalization framework that maps a given UML model into a formal specification language. Hydra automates this mapping process [3]. Specifically, we have created a UML-to-Promela formalization, supported by Hydra, tailored to the unique properties of embedded systems. This formalization maps objects to processes in Spin (*proctypes*) that exchange messages via *channels*. Nested and concurrent states are also formalized as processes. For the purposes of this paper, the formalization framework is configured to read UML 1.4 [2] models[1] specified in terms of XMI 1.1 [2] and generate Promela [7] specifications.

[1] CASE tool support for UML 1.5 and UML 2.0 is still limited.

To use the SMV model checker [23, 8], Tanuan and Atlee [5] have developed a set of rules to translate a UML model into SMV's specification language. Currently, there does not exist a tool that automatically translates UML models to SMV specifications. Therefore, we manually translate a UML model into an SMV specification using these rules. (We are extending Hydra to support this formalization.)

2.3 Step 3: Analyzing a UML Model

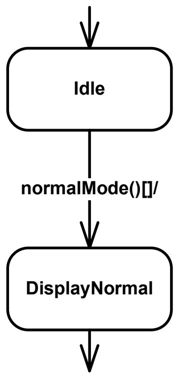
Next, the developer uses a model checker to analyze the formalized UML model for adherence to the previously specified property. If the model checker finds a violation of the property, then a violation trace is returned. The violation trace contains the sequence of steps performed by the system that lead to the violation.

3 Theseus Visualization Framework

The Theseus visualization framework, shown in the shaded region of the activity diagram in **Fig. 1**, supports visually interpreting the analysis results generated by model checkers in terms of the original UML diagrams. For example, **Fig. 2(a)** depicts a state diagram that has been analyzed for our adaptive light controller case study that will be described in detail in Section 4. **Fig. 2(b)** is an excerpt of the corresponding violation trace generated by Spin. From the trace files, Theseus extracts four types of dynamic behavior to animate: (1) A state is visited; (2) A transition is taken; (3) A message is sent; and (4) A message is received. The Theseus visualization framework comprises two components to depict this behavior: the Theseus trace processor and the Theseus visualization engine. The *Theseus visualization engine* takes the XML intermediate representation of the dynamic behavior from the trace output and the original UML model as inputs and produces the UML state diagram animations and UML sequence diagram generation. We describe the Theseus trace processor and visualization engine in more detail.

3.1 Theseus Trace Processor

The objective of the *Theseus trace processor* (depicted in **Fig. 1**) is to identify the dynamic behavior within the violation trace file and to specify this behavior in an intermediate XML representation. It comprises a parser, which must be constructed for each syntactically unique trace file format, and a translator. Note that different trace file formats will be generated by different model checkers or by the same model checker with different output options or different instrumentation. However, each parser *is* reusable across traces generated from the analysis of different UML models and/or different properties by the same model checker with the same output options selected. Each parser constructs an ASG (abstract syntax graph) representation of the dynamic behavior specified by the trace file. The translator then traverses the ASG and creates an intermediate XML representation of the dynamic behavior.



(a) State Diagram

```
6:    proc 17 (UserInterface) line 1680 "pan_in" (state 1)
      [goto Idle]
8:    proc 17 (UserInterface) line 1714 "pan_in" (state 50) [(1)]
      in state UserInterface.Idle
8:    proc 17 (UserInterface) line 1714 "pan_in" (state 51)
      [printf('in state UserInterface.Idle\\n')]
...
200:  proc 17 (UserInterface) line 1725 "pan_in" Recv normalMode
      <- queue 12 (UserInterface_q)
200:  proc 17 (UserInterface) line 1725 "pan_in" (state 65)
      [UserInterface_q?normalMode]
      Transition to UserInterface.DisplayNormal (evt:normalMode())
202:  proc 17 (UserInterface) line 1727 "pan_in" (state 67)
      [printf('Transition to UserInterface.DisplayNormal
      (evt:normalMode()) ')]
204:  proc 17 (UserInterface) line 1698 "pan_in" (state 27) [(1)]
      in state UserInterface.DisplayNormal
204:  proc 17 (UserInterface) line 1698 "pan_in" (state 28)
      [printf('in state UserInterface.DisplayNormal\\n')]
```

(b) Corresponding Violation Trace

Fig. 2. Sample State Diagram and Violation Trace

An excerpt of a violation trace generated by Spin is depicted in **Fig. 2(b)**. It contains information from four different sources: the UML model, the Promela specification, any instrumentation added by Hydra, and internal Spin information (e.g., line numbers, process number, Spin states, etc.). The Theseus parser extracts the information corresponding to the four dynamic behaviors of interest and represents it as an ASG. We give examples of each as follows:

1. **A UML state is visited:**
   `6: proc 17` **(UserInterface)** `line 1680 ‘‘pan_in’’ (state 1) [goto` **Idle**`]`
   The portion of the statement depicted in typewriter font specifies Spin internal information that is irrelevant for visualization purposes. Specifically, `6: proc 17` represent the execution step and internal Spin process number, respectively. `line 1680 ‘‘pan_in’’ (state 1)` are the line number within and the file name of the trace file, and the Spin internal state, respectively. This statement specifies that the UserInterface visits state Idle.
2. **A UML transition is taken:**
   `Transition to` **UserInterface.DisplayNormal** `(evt:`**normalMode**`()ˆ` **Display.showNormMes**`)`
   This statement is produced by the instrumentation (from Hydra) added to the Promela specification. Spin can provide this information, but only by activating specific flags to generate even more verbose and cumbersome output. Therefore, since we have the ability to extend Hydra, for convenience we have added instrumentation to obtain this information. This statement denotes that the UserInterface transitions to state DisplayNormal as a result of the **normalMode** event occurring. In addition, as a result of this transition being taken, the message **showNormMes** is sent to Display.

3. **A UML message is sent:**
   `201: proc 17 (`**UserInterface**`) line 1726 ‘‘pan_in’’ Send` **showNormMes** → `queue 13 (`**Display_q**`)`
   This statement specifies that UserInterface sends the message **showNormMes** to Display.
4. **A UML message is received:**
   `206: proc 11 (`**Display**`) line 1248 ‘‘pan_in’’ Recv` **showNormMes** ← `queue 13 (Display_q)`
   This statement specifies that Display receives the message **showNormMes**.

The Spin translator translates the ASG representation of the dynamic behavior generated by the parser into an XML intermediate format. Specifically, there is an intermediate XML specification for each of the four types of dynamic behavior. For example, **Fig. 3(a)** shows a sample XML element specifying that state Idle in class UserInterface is visited. **Fig. 3(b)** specifies that object Display sent a message named **showNormMes** to object UserInterface.

```
<Expression>
  <Process name="UserInterface"/>
  <Goto>
    <Read_location>
      <Process name="UserInterface"/>
      <State name="Idle"/>
    </Read_location>
  </Goto>
</Expression>
```

(a) Visited State

```
<Expression>
  <Process name="UserInterface"/>
  <Send_Message>
    <Message name="showNormMes"/>
    <End_Transition>
      <Queue name="Display"/>
    </End_Transition>
  </Send_Message>
</Expression>
```

(b) Sent Message

Fig. 3. Sample XML Elements

3.2 Visualization Engine

The visualization engine has been implemented in the ArgoUML [19] CASE tool as a plugin. ArgoUML was selected because of its open source application programming interface (API) that allows the creation of plugins. Theseus provides two animation options: automatic playback and incremental playback. Automatic playback animates the complete violation trace; whereas, incremental playback animates the animation trace in a stepwise fashion (single or multi-step). The multi-step option is useful when there are a large number of steps in the violation trace and the developer suspects the first several steps may not be relevant to the violation. After skipping to a specific step, the developer is able to automatically play the remaining steps, or incrementally play the next step.

Theseus provides two mechanisms for visualizing violation traces on the UML model, *state diagram animation* and *sequence diagram generation*. Specifically, state diagram animation depicts that a state is visited (colored red when visited and turns yellow upon departure) and that a transition fires (in red). The generated sequence diagram is animated to depict that a message is sent (arrow in red) and received (arrow in blue). As such, Theseus depicts all four types of dynamic behavior useful for understanding a violation trace.

Both state diagram and sequence diagram animations help a developer to better understand the cause for a property violation. While the state diagram animation is better suited for understanding the behavior of an individual object, the generated sequence diagram helps a developer to understand the context for a property violation in terms of object interaction. Note that typically a UML diagram may have several state diagrams, each of which represents the behavior of a particular object in the system. Currently, Theseus displays the state diagram of a particular object, depending on the part of the counterexample being traversed. As events and messages communicate among objects, the corresponding object's state diagram is displayed. In future versions, we plan to display more than one state diagram at a time in addition to the sequence diagram.

4 Case Study

This section describes an industrial case study we performed to validate our visualization framework. Specifically, object analysis patterns were used to create a UML model of an embedded system application, Hydra generated a formal specification of the UML model, Spin verified critical system properties specified with SPIDER, and Theseus visualized the analysis results in terms of the original UML diagrams. Due to space constraints, we do not include a case study for the SMV visualization, but a description may be found in [24].

4.1 Adaptive Light Control System

The adaptive light control system (ALCS) is responsible for moderating the lights in a room. A class diagram depicting the structure of this system is depicted in **Fig. 4**. The class attributes and operations have been elided due to space constraints.

The primary function of the ALCS is to ensure that if the room is occupied, then the room is sufficiently illuminated, either by natural light or by the lamps. The ALCS comprises a switch for manually turning on the lights, a display for communicating messages to a user, a motion sensor for detecting that the room is occupied, a brightness sensor for detecting the current illumination level of the room, and a dimmer that controls the brightness of the lamps. The *Controller Decompose*, *Actuator-Sensor*, *User Interface*, *Computing Component*, *Fault Handling*, and *Detector-Corrector* object analysis patterns have been used in the specification of the structure and behavior of the ALCS. For additional details about these patterns, please refer to [15]. Note, **Fig. 2(a)** describes a portion of the behavior of the UserInterface.

4.2 Property Specification and Analysis

We analyzed the UML model for the ALCS using the Spin model checker. First, we used Hydra to translate an XMI representation of the UML model into

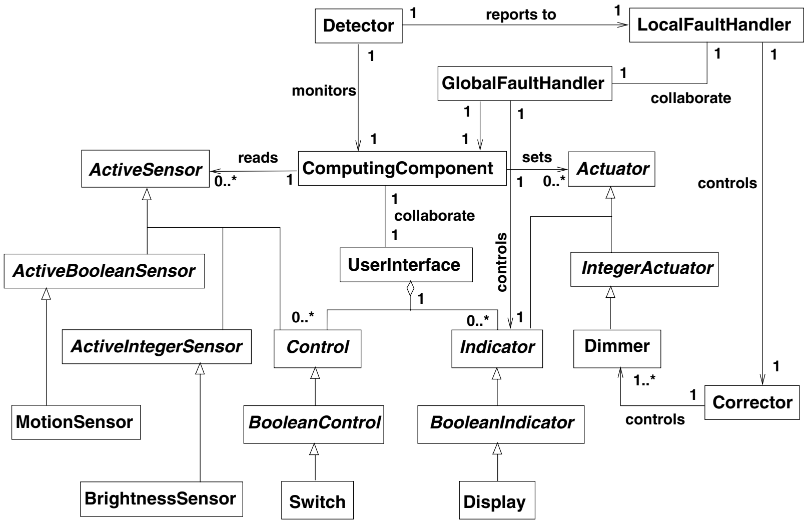


Fig. 4. Class Diagram of the ALCS

Promela. Second, we used Spider to formally specify properties to be satisfied by the model. For example, using Spider, we created the following natural language property:

> "Globally, it is always the case that if the initialization has succeeded, then eventually the display shows the initialization succeeded message." (1)

Spider then extracts UML model elements from the ALCS model to instantiate the free-form text the initialization has succeeded and the display shows the initialization has succeeded message with model-specific elements. The initialization in the ALCS has succeeded if the lightStatus of the ComputingComponent is set to value 1. Therefore, the initialization has succeeded is replaced with `ComputingComponent.lightStatus=1`. Similarly, the text the displays shows the initialization succeeded message is replaced with `call(Display.showNormMes())` to denote that the message *showNormMes()* of the Display is called. Thus, we obtain the following instantiated natural language property:

> "Globally, it is always the case that if `ComputingComponent.lightStatus=1`, then eventually `call(Display.showNormalMes())`." (2)

From this specification, Spider automatically creates the formal specification of the property in LTL:

$$\Box(\texttt{(ComputingComponent.lightStatus=1)} \rightarrow \Diamond\texttt{(call(Display.showNormMes()))}) \tag{3}$$

At this point, Spider invokes Spin with the Promela model of the ALCS and the LTL property. In this case, model checking detected a violation and Spin generated a violation trace.

4.3 Property Visualization

Theseus processed the violation trace and visualized the counterexample in terms of the original UML state diagrams and a sequence diagram. A screen shot of a state diagram animated to depict one step of the violation trace is depicted in **Fig. 5**, where the key thing to note is the different colors of the states and the transitions. In this case, we are viewing the state diagram for the UserInterface object. (The intent of these figures is not to read the individual names of states or transitions, but to note the color changes – or the levels of shading in gray scale.) A screen shot of the sequence diagram generated by Theseus depicts the violation trace shown in **Fig. 6**. Using the Theseus visualizations of the violation path, we were able to locate the source of the error and revise the UML model accordingly. Rerunning the overall process yielded no further violations. Without Theseus, we would be forced to understand the syntax and semantics of the trace output, determine the relationship between the output and the UML model, and then locate the corresponding error within the UML model.

5 Related Work

Numerous CASE tools [10, 11, 12, 13, 14] provide visualization support for UML model simulation. To the best of our knowledge, they do not support the visualization of violation traces gathered during model checking analysis in terms of the original UML diagrams. Most formal analysis tools, in contrast, offer visualization capabilities in terms of the analysis models, such as Spin [7] and UPPAAL [25]. However, this visualization is on the level of the description language of the formal analysis tool and not at a more abstract level, such as a UML model.

Other tools visualize analysis results from model checkers in terms of UML. vUML [4] translates UML diagrams into Promela and uses Spin for analysis purposes. Violation traces revealed by formal analysis may be displayed in terms of UML sequence diagrams. To keep the model checking process transparent, vUML focuses on the analysis of more general properties, such as deadlocks and livelocks. Differing from our work, vUML only supports the translation of UML models to Promela, does not support the construction of property specification in terms of natural language or formal specification languages, and does not offer state diagram animation capabilities. MOCES [26] translates Statemate [11] state charts into Promela. The semantics of the Statemate state charts differs from the semantics for UML state diagrams [27]. In addition, MOCES only supports the analysis of a single state chart, while our tool analyzes behavior captured in a collection of collaborating state machines. Hugo/RT [28] supports the analysis of UML diagrams using Spin or UPPAAL [25]. In addition, Hugo/RT can translate a violation trace produced by these analysis tools

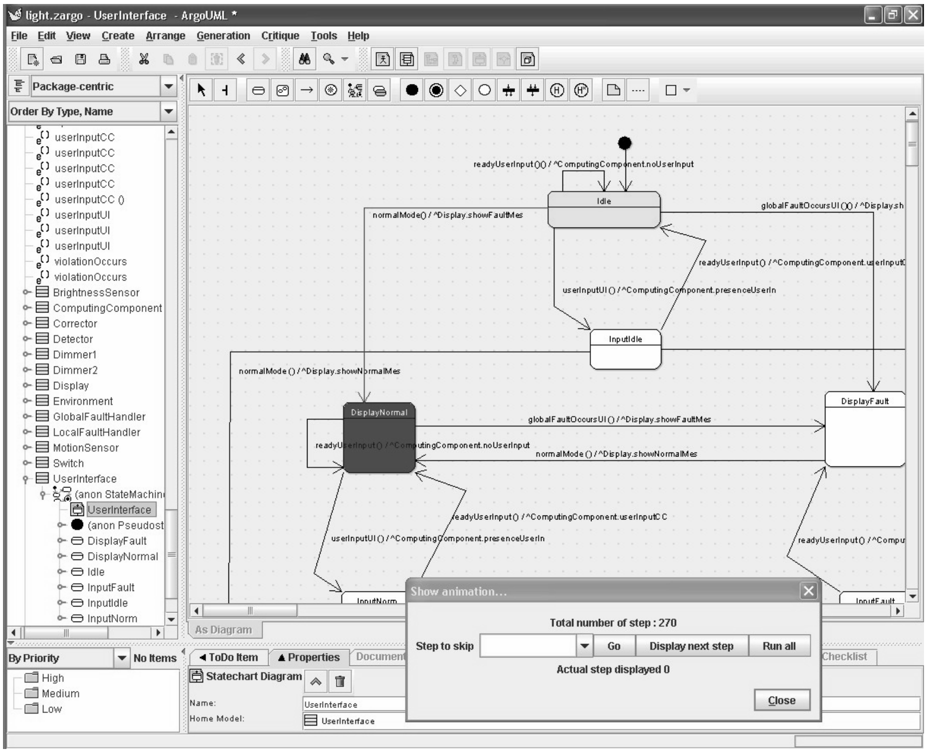

light.zargo - UserInterface - ArgoUML *
File Edit View Create Arrange Generation Critique Tools Help
Package-centric
Order By Type, Name
Idle
InputIdle
DisplayNormal
DisplayFault
InputNorm
InputFault
readyUserInput()() / ^ComputingComponent.noUserInput
normalMode() / ^Display.showFaultMes
globalFaultOccursUI()() / ^Display.sh
readyUserInput() / ^ComputingComponent.userInput
userInputUI() / ^ComputingComponent.presenceUserIn
normalMode() / ^Display.showNormalMes
globalFaultOccursUI() / ^Display.showFaultMes
readyUserInput() / ^ComputingComponent.userInputCC
Show animation...
Total number of step : 270
Step to skip
Go
Display next step
Run all
Actual step displayed 0
Close
By Priority
No Items
High
Medium
Low
ToDo Item
Properties
Statechart Diagram
Name:
UserInterface
Home Model:
As Diagram

Fig. 5. Theseus Animation of Adaptive Light Controller Violation Path

to a representation in terms of UML elements. However, Hugo/RT provides a proprietary textual UML representation and does not interactively display the violation trace in terms of a graphical UML representation in a CASE tool. In summary, none of the aforementioned tools combines the capability of displaying analysis results in terms of UML sequence and state diagrams and the customizability towards numerous formal analysis tools.

6 Conclusions

This paper has described a generic visualization framework that provides a critical link in a roundtrip-engineering process for modeling and analyzing embedded systems. The prototype of this visualization framework, offers three key benefits to UML modelers who want to model check their UML diagrams. First, Theseus supports modelers who are not proficient in interpreting the verbose and often cryptic analysis results generated by model checkers, by locating the source of the error identified by the violation trace. Theseus visually animates the violation trace on the UML state diagrams and a generated sequence diagram. Second, Theseus is extensible to other formal analysis tools beyond the

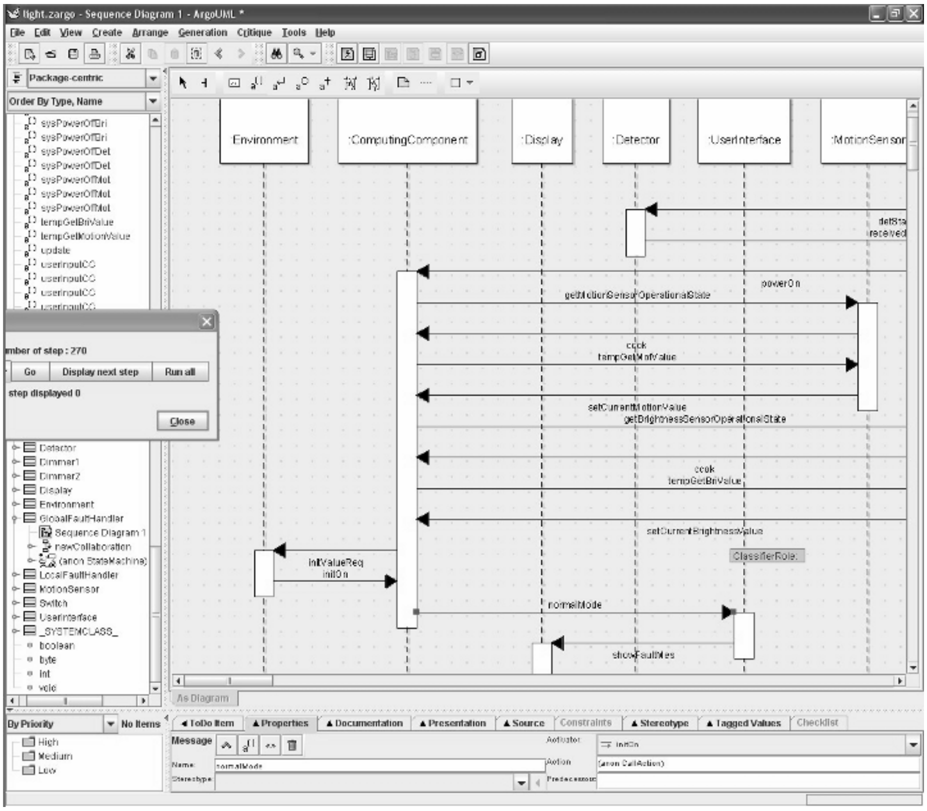


Fig. 6. Theseus Generated Sequence Diagram depiction of Adaptive Light Controller Violation Path

ones mentioned in this paper. To extend Theseus to visualize output from other analysis tools, a specific Theseus trace processor needs to be constructed. The parser of the trace processor depends on the formalization rules (i.e., the rules for mapping UML to the target specification language of a given analysis tool) and the violation trace output options used in the model checker (including any instrumentation added to the trace output). The translator of the trace processor, however, depends only on the formalization rules, and is potentially reusable for different violation trace output options. Currently, we have developed trace processors that support output generated by the SMV and Spin model checkers. Independent of the model checker, the formalization rules, and the output options, the Theseus visualization engine is reusable across the trace output for different state-based analysis tools. Third, Theseus completes the roundtrip modeling and analysis process for embedded systems by enabling a developer to automatically formalize a UML model, specify natural language properties that the model must satisfy, analyze the model for adherence to these properties, and visualize property violations in terms of the original UML diagrams.

Future work will include applying Theseus to additional case studies and extending Theseus in different directions. First, we are extending Theseus to view multiple state diagrams side-by-side during animation. Additionally, we are investigating how to extend the Theseus framework to visualize the analysis results from complementary model checkers, such as the real-time model checkers Kronos [29] and UPPAAL [25]. Finally, we are exploring a more seamless integration between the tools and the steps of our roundtrip-engineering process for modeling and analysis. The biggest challenge has been the vendor-specific differences in the implementation of the standard for data interchange between tools and third parties.

Bibliography

[1] Douglass, B.P.: Real-Time Design Patterns. Addison-Wesley (2003)
[2] Object Management Group: `http://www.omg.org`.
[3] McUmber, W.E., Cheng, B.H.C.: A general framework for formalizing UML with formal languages. In: Proc. of the IEEE Int. Conf. on Software Engineering (ICSE01), Toronto, Canada (2001)
[4] Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models. In: Proc. of the 14th IEEE Int. Conf. on Automated Software Engineering, Washington, DC (1999)
[5] Tanuan, M.C.: Automated Analysis of Unifed Modeling Language (UML) Specifications. Master's thesis, University of Waterloo, Canada (2001)
[6] Inverardi, P., Muccini H., Pelliccione P.: CHARMY: An Extensible Tool for Architectural Analysis. In: ESEC/FSE-13: Proc. of the 10th European Software Engineering Conf. held jointly with 13th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering. (2005)
[7] Holzmann, G.: The Spin Model Checker. (2003)
[8] McMillan, K.L.: Getting started with SMV (1999)
[9] IBM: Rational Rose XDE Developer. `http://www-306.ibm.com/software/awdtools/developer/rosexde/` (2005)
[10] Telelogic: ObjectGEODE. `http://www.telelogic.com/` (2005)
[11] I-logix: `http://www.ilogix.com/` (2005)
[12] ARTiSAN Software: Real-time Studio. `http://www.artisansw.com` (2005)
[13] Ho, W.M., Jézéquel, J.M., Guennec, A.L., Pennaneac, F.: UMLAUT: an extendible UML transformation framework. In: Proc. of the Automated Software Engineering Conf., Florida (1999)
[14] Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proc. of the 22nd Int. Conf. on Software Engineering, New York, NY, USA, ACM Press (2000) 742–745
[15] Konrad, S., Cheng, B.H.C., Campbell, L.A.: Object Analysis Patterns for Embedded Systems. IEEE Trans. on Software Engineering **30**(12) (2004) 970 – 992
[16] Konrad, S., Cheng, B.H.C.: Facilitating the construction of specification pattern-based properties. In: Proc. of the IEEE Int. Requirements Engineering Conf. (RE05), Paris, France (2005)
[17] Konrad, S., Cheng, B.H.C.: Automated analysis of natural language properties for UML models. In Bruel, J.M., ed.: Satellite Events at the MoDELS 2005 Conf.: MoDELS 2005 Int. Workshops, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers. Volume 3844 of Lecture Notes in Computer Science., Springer-Verlag GmbH (2006) 48–57

[18] Manna, Z., Pnueli., A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc. (1992)

[19] Tigris.org: ArgoUML: The project home. `http://argouml.tigris.org` (2005)

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

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

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

[23] McMillan, K.L.: Symbolic Model Checking. PhD thesis, Carnegie Mellon University (1993)

[24] Kamdoum, S.: Facilitating the roundtrip engineering of model-driven software architecture. Master's thesis, Michigan State University (2006)

[25] Pettersson, P., Larsen., K.G.: Uppaal2k. Bulletin of the European Association for Theoretical Computer Science **70** (2000) 40–44

[26] Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing statecharts in promela/spin. In: WIFT '98: Proc. of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques (1998) 90

[27] Crane, M.L., Dingel, J.: UML vs. classical vs. Rhapsody statecharts: Not all models are created equal. In: Proc. of the ACM/IEEE 8th Int. Conf. on Model Driven Engineering Languages and Systems. (2005)

[28] Knapp, A., Merz, S., Rauh, C.: Model checking timed UML state machines and collaborations. In Damm, W., Olderog, E.R., eds.: 7th Int. Symposium 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

[29] Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: a model-checking tool for real-time systems In: Proc. of the 10th Conference on Computer-Aided Verification. (1998)