A Toolchain for the Detection of Structural and Behavioral Latent System Properties*

Adam C. Jensen, Betty H.C. Cheng, Heather J. Goldsby

Michigan State University, East Lansing MI 48824, USA
{acj,chengb,hjg}@cse.msu.edu

Edward C. Nelson

Ford Research and Advanced Engineering, Dearborn, MI 48121, USA
enelson7@ford.com

Abstract. The cost to repair a requirements-based defect in software-based systems increases substantially with each successive phase of the software lifecycle in which the error is allowed to propagate. While tools exist to facilitate early detection of design flaws, such tools do not detect flaws in system requirements, thus allowing such flaws to propagate into system design and implementation. This paper describes an experience report using a toolchain that supports structural and behavioral analysis of UML state diagrams that is not currently available in commercial UML modeling tools. With the toolchain, models can be incrementally and systematically improved through syntax-based analysis, type checking, and detection of latent behavioral system properties, including feature interactions. This paper demonstrates use of the toolchain on an industry-provided model of onboard electronics for an automotive application.

Keywords: requirements engineering, UML, latent properties, model checking

1 Introduction

In software development, the cost to repair a defect increases substantially with each successive phase of the software lifecycle [1, 2]. When a defect is allowed to propagate into the design and implementation phases, the number of artifacts (e.g., models and documentation) that are affected by it also increases. Typically, during the requirements phase the system's stakeholders describe the key needs and problems that the system-to-be should address, usually using natural

* This work has been supported in part by NSF grants CCF-0541131, IIP-0700329, CCF-0750787, CCF-0820220, DBI-0939454, CNS-0854931, Army Research Office grant W911NF-08-1-0495, Ford Motor Company, and a Quality Fund Program grant from Michigan State University.

language. As a means to clarify and refine requirements that have been expressed in natural language, developers construct *domain models* that identify the key elements of the system and their relationships to one another, as well as their relationships to external elements. In order to better understand the required behavior, developers often create prototypes or state-based representations based on the domain model. While simulations and executable prototypes enable *validation* of requirements, it is equally important to be able to *verify* requirements to identify inconsistencies, (invariant) property violations, etc. Thus, there is a need for tools that identify errors in requirements specifications based on analysis of early prototype models. This paper presents a toolchain that facilitates the detection of syntactic and semantic errors in state-based diagrams and also identifies properties that specify *latent behavior*, the unspecified and potentially unwanted behavior of the model.

Many tools have been developed to support model-driven engineering of software systems. Tools such as ArgoUML [3], Rational Software Architect [4], and Microsoft Visio [5] support visual modeling of software designs via the Unified Modeling Language (UML). IBM Rational Rhapsody [6] supports UML modeling as well as code generation and many consistency tests to ensure that the system under development is free of syntax errors. However, none of these tools performs syntax or type checking on state transition expressions in state diagrams. Particularly for applications involving complex logic and system behavior (e.g., embedded systems), transitions may contain complex guards and action statements that often define the core functionality of the system being modeled. Thus, tools that treat the transition expressions as uninterpreted strings allow subtle errors to propagate into the source code that is generated from the model, particularly in the context of model-driven engineering. Furthermore, while tools such as Rhapsody provide traceability from requirements to source code, to the best of the authors' knowledge no existing commercial or research tools provide the comprehensive automated identification of the collection of different types of errors covered by our toolchain for UML models.

This paper describes an experience report from using a newly-developed toolchain that supports syntax and type checking as well as detection of latent system properties. After requirements have been elicited for an embedded system, developers often build a domain model using class diagram syntax that describes the key elements of the system (including physical elements, such as sensors and actuators, and software elements, such as controllers) and elements in the environment with which the system interacts. A state diagram is created for each key element, resulting in a collection of interacting state diagrams. While such diagrams are useful for refining system requirements and may be used during the design phase, there is limited tool support for detecting errors in syntax and semantics, and to our knowledge there is no tool support for automatically identifying latent properties. The proposed toolchain has two key advantages over current approaches. First, all state transition expressions are parsed and type-checked, thus identifying many errors that existing tools do not address until the code generation phase. Second, automated detection of latent properties enables system developers to identify so-called *blind spots* in system requirements. Blind spots are missing or incomplete requirements that are overlooked by requirements engineers, and they are often discovered only after the system has been partially implemented or, worse yet, deployed to the field. By identifying these errors early in the development process and suggesting resolution strategies when possible, the proposed toolchain minimizes the number of subtle design defects and the cost of redesigning the system to correct the defects.

The proposed toolchain comprises three main tools: Cyclops, a model preprocessor that identifies common syntax and semantics errors in behavioral models specified in XMI (XML Metadata Interchange) format; Hydra, a tool for translating UML behavioral models into Promela, the formal language for the SPIN model checker [7]; and Marple, a tool for automatically generating properties that are satisfied by the model and may represent latent and potentially erroneous behavior. We apply this toolchain to an industrial software system from the automotive embedded systems domain. The software system comprises three subsystems: Lighting, Power Management, and Windshield Wipers. The Lighting subsystem handles all functionality related to interior lamps, headlights, and tail lights. The Power Management subsystem monitors and controls the ignition status, vehicle speed, door statuses, battery status, and other electronic features. The Windshield Wipers subsystem controls the movement and speed of the windshield wipers. The subsystems are sophisticated and interact with one another at run time, thus creating the potential for errors in modeling semantics, unintended behavior that spans multiple subsystems, and feature interactions.

Based on feedback from the developer of the model, it is clear that several of the detected errors would have been very difficult and time-consuming to detect and resolve without the use of the toolchain. The remainder of the paper is organized as follows. In Section 2, we discuss background concepts. We present the software model that was studied in this work in Section 3. Next, we describe the process of using the toolchain in Section 4. Section 5 describes related work. Our experience of applying the toolchain to an automotive embedded systems model is presented in Section 6. We discuss the results and consequences of applying the toolchain in Section 7. Finally, we present our conclusions and discuss future work in Section 8.

2 Background

In this section, we discuss background concepts and enabling technologies that support the proposed toolchain, including the Unified Modeling Language, the SPIN model checker, evolutionary computation, and novelty search. These enabling technologies are presented according to the tool(s) that leverage their capabilities.

2.1 Cyclops and Hydra

Cyclops and Hydra have been developed to support the construction of models in the Unified Modeling Language (UML) [8], the *de facto* standard in object-oriented software modeling. They enable developers to perform extensive error checking on UML models that describe system prototypes and support the translation of UML state diagrams into Promela for analysis with the SPIN model checker.

Unified Modeling Language. The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used for modeling object-oriented software. It comprises several types of diagram notations, including support for class diagrams, interaction diagrams, state machine diagrams, and others. A UML model may contain many different diagrams that describe different views of the same system. For the purposes of this paper, we assume the use of UML version 1.5 and focus on state machine diagrams. A state machine diagram (hereafter, "state diagram") describes the various states in which a system can be and the transitions between the states. Visually, a state diagram comprises rounded rectangles (representing states) and lines with arrows that indicate transitions between states. The lines are annotated with optional guards and trigger events that denote the conditions that enable a transition and the actions that are generated as a result of the transition, respectively. In this study, we use a domain model (expressed in terms of a class diagram notation) to provide the context and vocabulary for the state diagrams.

SPIN Model Checker. The SPIN model checker [7] is a tool for exhaustively verifying state-based models. It takes a model expressed in Promela and produces a model checker in C code. SPIN uses nondeterministic automata to check properties expressed in Linear Temporal Logic and performs exhaustive analysis of a system's state space in order to identify undesirable system behaviors. It was originally developed to formally analyze telecommunications protocols, but in recent years it has also been used to analyze distributed systems [9, 10].

2.2 Marple

Marple is a tool that automatically discovers latent properties in UML state diagrams. It leverages novelty search, an evolutionary search technique, and formal model analysis to generate a list of properties that describe the behavior specified by the model.

Evolutionary Computation. Evolutionary computation (EC) is a biologically-inspired family of techniques for exploring large solution spaces using concepts such as mutation and selection [11]. EC is effective for finding solutions to problems that have large solution spaces that cannot be exhaustively explored in a reasonable amount of time. It begins with a large population of randomly-generated individuals. Each individual is evaluated to determine its fitness for a given task. Next, an EC algorithm probabilistically selects a set of individuals that will represent the next generation. Each selected individual is probabilistically mutated, thus introducing diversity into the population. This process of selection, mutation, and evaluation continues until a fixed number of generations have passed or an optimal solution (if one exists) has been found.

Novelty Search. One EC technique, known as *novelty search* [12], replaces the explicit fitness computation with a novelty function that measures how different each individual is from other individuals in the population and in an archive of previous individuals. Novelty search then selects individuals whose behavior is the most distant (i.e., the most novel), thus increasing the diversity in the population and exploring the solution space more efficiently than a random search. The specific measure of distance between individuals varies with the problem being solved, but a Euclidean distance is typically used when the behavior of an individual can be mapped to a numerical vector.

3 Body Subsystem Model

In this section, we describe the Body Subsystem model that was used in this study. The model describes and simulates embedded devices that control the electronic subsystems of a modern passenger automobile and was created for the purposes of requirements elicitation and analysis.[1] The subsystems of the model include interior and exterior lighting, power management, and windshield wiper control. While the onboard electronics involves several more subsystems, these three were selected because they exhibit known, intended interactions. One of our objectives was to investigate whether the subsystems also exhibit unknown interactions. The remainder of this section provides a brief description of each subsystem under study.

3.1 Lighting Subsystem

The Lighting subsystem comprises 16 classes and is responsible for managing interior lights, including map, vanity, trunk, and under-hood lamps; and exterior lights, including head lights (low- and high-beam) and tail lights. The subsystem also contains classes that monitor the intensity of ambient light in order to control day time running lights and activate the vehicle's head lights and tail lights for night time driving.

3.2 Power Management Subsystem

The Power Management subsystem comprises 25 classes and is responsible for monitoring ignition status, sleep mode status, battery voltage, and commands from remote key fobs. The subsystem responds to events such as the insertion of an ignition key, exceeding vehicle speed thresholds, and the firing of timers.

3.3 Windshield Wiper Subsystem

The Windshield Wiper subsystem comprises eight classes and is responsible for controlling wiper behavior. The classes represent hardware and software ranging from the low-level motor controller, the washer fluid pump, and a stall sensor that turns off the wiper motor if it detects that the wipers are not moving.

[1] The model was developed by the industrial partner as an example of an industrial-strength model with representative system elements and behavior. The model does not contain any proprietary or specific configuration parameters of a deployed vehicle.

4 Process

In this section, we provide an overview of the process that was used to apply the toolchain to the Body Subsystem model. A data flow diagram for the process is shown in Figure 1. The process begins with a system model in XMI (XML Metadata Interchange) format. In this case, Rhapsody was used by our industrial collaborators to create the system model because of its support for requirements traceability, code generation capabilities, and support for state-based modeling.

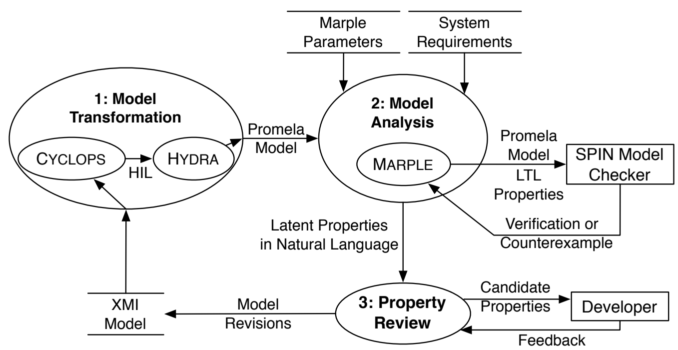

Fig. 1. Data Flow Diagram

4.1 Model Transformation

First, the XMI model is given to step **1: Model Transformation**. The Cyclops tool takes the XMI model and checks for common syntax and semantics errors. For example, it parses and checks each state transition expressions to ensure that they are well-formed and do not refer to undeclared classes, attributes, or operations. Cyclops produces specific error messages that indicate the nature of any errors that are discovered, and it makes suggestions when appropriate (e.g., when an attribute from another class is referenced as though it were declared in the current class). Cyclops supports an iterative process of analysis, detection of errors, and model correction. This incremental error-correction cycle is shorter and more interactive than comparable techniques available in commercial tools. For example, using code generation to detect syntactic and semantic errors in a model would require at least one additional step for compilation and linking compared to our toolchain. Once the errors detected by Cyclops are resolved, it translates the XMI model into the Hydra Intermediate Language (HIL) that can then be processed by the Hydra tool.

Hydra is a model translator initially developed by McUmber [13]. It takes a model in HIL format and produces an equivalent model in Promela (the PROcess MEta Language). Promela is a formal logic language that was developed to support analysis and exhaustive checking of concurrent systems of communicating processes [14]. Promela models are checked using the SPIN model checker [7], which identifies livelocks, deadlocks, error conditions, and other undesirable behavior. It also has support for verifying arbitrary properties specified using Linear Temporal Logic (LTL) [15].

4.2 Model Analysis

Goldsby and Cheng developed Marple, a novelty-search tool for automatically discovering properties that represented the behavior of UML models. Specifically, a property may specify a known system requirement or, more interestingly, an unknown latent behavior of the model [16]. As part of the **2: Model Analysis** step, Marple accepts the Promela model generated by Hydra and a set of parameters as input. Marple parameters include the number of properties that should be returned, the size of the population that the novelty search algorithm should use, and the number of distinct classes (i.e., domain elements) that are mentioned in each property. The parameters may be tuned by the system developer according to the model being analyzed and the number of results that are desired.

Within Marple, each individual is an LTL property created by instantiating one the five most commonly occurring LTL specification patterns [17] with model specific elements provided as part of the parameters. Each property is evaluated using the SPIN [7] model checker. To assess the novelty of a property that holds for the model, Marple compares the state space of the shortest path that satisfies the property to the state space of other properties. If the property visits a previously unexplored region of the state space, then the property is considered more novel, and thus more fit, than a property that visits states within a well-explored region. New properties are compared to other properties within the current generation and within the archive of previously generated properties. By including the properties in the archive in the comparison, the novelty search algorithm is able to remember the portions of the solution space that it has explored previously, thus ensuring that the algorithm does not stagnate or become stuck in a suboptimal portion of the space.

As output, Marple produces a set of LTL properties that are presented to the developer in natural language for readability purposes. To enable this natural language property representation, we use a component of SPIDER [18], a specification pattern instantiator and analysis tool to translate between LTL properties and natural language [18].

4.3 Property Review

Finally, in step **3: Property Review**, the latent properties discovered by Marple are presented in natural language to the system developer for review. If a given property is desirable, then the developer may consider adding it to the list of explicit system requirements. If the property is undesirable, however, action must be taken to ensure that the property does not continue to hold. For example, the developer might examine the state diagrams for the classes that are mentioned in the property. If an error is discovered in the diagrams, then the model is revised and the toolchain is restarted at step **1**.

5 Related Work

As stated earlier, many commercial tools support the creation of UML models, syntax checking, simulation, and code generation capabilities. However, they do not support the automated detection of the full suite of syntactic and semantic error checks for state-based diagrams that we describe in this paper. Additionally, they do not support the identification of latent properties satisfied by the model. In this section, we explore research tools that have been created to address these two challenges.

5.1 Consistency Checking Among UML Class and State Diagrams

One key challenge that arises as the result of using multiple diagrams to provide different views of the same system is maintaining consistency among these different representations. As a result, researchers have developed a number of approaches to support various aspects of consistency checking among UML models (e.g., [19–25]). The toolchain described by this paper automatically detects inconsistencies in the syntax and semantics of UML class and state diagrams created as part of the late requirements engineering phase of development. Thus, we focus our attention on approaches that examine consistency among these two diagram types. Simmonds *et al.* use rules presented in terms of description logic [26], a subset of first order predicate logic, to identify inconsistencies among UML class, sequence, and state diagrams during the design phase [24]. However, their approach does not check that the transitions within the state diagram use viable elements from the class diagram. Gomaa *et al.* present an approach to checking the consistency among use case diagrams, class diagrams, sequence diagrams, and state diagrams [22]. Their manual approach involves specifying consistency checking rules among the various types of diagrams, including class and state diagrams. Egyed proposes a Rational Rose plugin that can be used to detect and resolve inconsistencies that arise within UML models at the design phase [19]. Their approach relies upon the specification of consistency rules, which are periodically evaluated. To the best of our knowledge, these consistency rules can detect whether elements of the state diagrams are consistent with those that appear in the class diagram, but do not detect subtle errors, such as assignments that occur within transition guards. Schwarzl and Peischl propose an approach to statically analyzing state diagrams for syntax, existence, data type, communication, non-determinism, and transition hiding errors [23]. As part of this process, the transitions on the state diagrams are checked for well-formedness. The set of syntactical and semantic errors that they detect is a subset of the errors that Cyclops detects. However, the behavioral errors that they detect (e.g., deadlock conditions and circular messaging dependencies) are complementary to errors detected by the approach presented in this paper.

5.2 Detection of Latent Properties

Several approaches generate temporal logic properties that specify the behavior of systems [27–31]. Because the objective of our approach is to automatically identify obscure latent properties that might not otherwise be discovered, we focus on how the approaches blend developer knowledge and automation to identify properties. Perracotta [31] is a dynamic inference approach that infers properties from imperfect execution traces, which have been generated by running the program code. To produce these execution traces, the developer must instrument the program to monitor events and states of interest; these are used to form the possible propositions. Perracotta then creates properties by instantiating eight variations of the temporal logic response pattern with the propositions. Weimer and Necula proposed a static inference approach [30], which analyzes program text and generates properties. These properties specify potentially erroneous behavior of the error-handling portions of the source code. Lastly, Chang *et al.* [28] proposed a dynamic inference approach that generates properties from program event traces. The program traces are created during the execution of the program and track developer-specified events. Chang's approach involves refining the inference templates built using the Propel patterns [32] to eliminate properties that are not satisfied by the program's event traces.

These approaches differ from our toolchain-based approach in two key ways. First, they focus on automatically generating properties that describe the behavior of the *code*, rather than the model. As such, the cost of correcting errors in the later development phase is likely to be more expensive. Second, in general, these approaches rely on the developer to select portions of the code to explore for properties. This limits the ability of the approaches to discover properties that represent unwanted latent behavior in blind spots. These notable differences mean that our approach can be used in a complementary fashion. Specifically, as part of the model-driven development process our toolchain can be used to automatically discover properties that may represent unwanted latent behavior within the UML model. Once the UML model has been translated to code, the other approaches could be used to ensure that no errors have been introduced.

6 Applying the Toolchain

This section describes our experience of applying the proposed toolchain to the Body Subsystem model that was presented in Section 3. We present the types of errors that were discovered, the mitigation strategy that was used for each error, and the consequences of correcting the error. For clarity, we present the errors according to the stages of the toolchain. That is, we begin with a discussion of syntax and consistency errors that Cyclops detected. Next, we discuss the errors in types and semantics that Cyclops also detected. Finally, we describe how the model was translated into the Promela language and discuss the latent properties that Marple discovered.

6.1 Preliminaries

The model comprises class diagrams, sequence diagrams, and state diagrams, thus providing a rich domain vocabulary (i.e., class, operation, and attribute names) as well as a complete set of states and transitions that represent the behavior of the system-to-be. The Body Subsystem model contains 52 classes, 37 state diagrams, 255 states (including composite states), and 400 state transitions. There are fewer state diagrams than classes because several of the classes are abstract superclasses or static classes that serve as structures. The model generated approximately 38,000 lines of C++ code. This code was intended to provide a means to execute the requirements; it is not intended to be sufficiently detailed to contain platform-specific or implementation details.

6.2 Phase I: Syntax and Consistency Check (Cyclops)

We begin by applying Cyclops to the model, which comprises class and state diagrams. Cyclops performs a battery of checks on the input model before it is passed to HYDRA to be translated into Promela. It examines each class, attribute, and operation reference and verifies that the referenced element exists. CYCLOPS also checks for unmatched or missing parentheses, missing semicolons between action statements, and ensures that attributes and operations do not have the same name as their owning class. It also ensures that each state transition expression is well-formed. CYCLOPS identified a wide range of errors in our model, including references to undeclared variables and typographical mistakes. The categories of errors that were discovered, and their frequency of occurrence, are shown on the left-hand side of Table 1.

Phase I Error Description	**Frequency**	**Phase II Error Description**	**Frequency**
Transition expression syntax	12	Misspelled identifier name	18
"==" operator in action list	9	Undefined variable	32
"=" operator in guard	4	Undefined operation	4
Missing/unmatched parentheses	16	Undefined enum. type	11
Missing semicolons	19	Incompatible types	7
Name collision	2	Attribute used as event	2

Table 1. Phase I and II Errors

Error Mitigation. Defects that are discovered during Phase I are typically inconsistencies that result from typographical errors. Automated tools cannot make reliable suggestions for resolving most defects of this type, and therefore CYCLOPS must rely on software engineers who are familiar with the model to correct the problem. Once each defect has been corrected, the revised model is given again as input to CYCLOPS, and the Phase I analysis is reapplied. It takes less than one second to parse and check the Body Subsystem, thus providing an interactive experience. This incremental defect resolution process proceeds until no further syntax errors are found in the model.

6.3 Phase II: Semantics and Type Check (CYCLOPS)

Next, we used CYCLOPS to check the semantics of each state transition in the model's state diagrams. CYCLOPS ensures, for example, that each reference to an attribute, operation, or class is valid with respect to the model being analyzed, using the domain model as a point of reference. Furthermore, CYCLOPS verifies that boolean comparisons and assignments are between compatible data types. The errors detected in this phase are shown on the right-hand side of Table 1.

Error Mitigation. Defects that are discovered during Phase II are more subtle, and therefore more difficult to detect, than those discovered during Phase I. The primary focus of Phase II is on parsing and verifying the contents of state transition expressions. A state transition expression specifies the conditions under which the modeled system will move from the current state to the next state and what actions (e.g., variable assignments or calls to operations) will be taken as a result of the transition. Each expression comprises an optional triggering event, a set of expressions that form a *guard*, and a set of actions to perform in the following format: `event[guard]/action-list` .

Errors in state transition expressions can be difficult to detect by visual inspection. For example, it is easy to overlook an assignment operator that was mistyped as an equality operator. Such an error still produces valid, executable code in many programming languages that are used for embedded systems (e.g., C). However, there is a disconnect between the intent of the code and its actual behavior when the system is executed, thus making this class of subtle defects potentially very serious.

6.4 Phase III: Model Translation

In the third phase, the model is free of syntactic and semantic errors and is ready to be translated by Cyclops and Hydra into the formal language Promela. Cyclops begins by translating the model into the Hydra Intermediate Language (HIL). This intermediate step enables us to build new front-end translators for successive versions of XMI, whose formats evolve over time, without needing to modify the core translation code in Hydra. Next, Hydra translates the HIL code into Promela. By constructing an equivalent model in Promela, we are able to conduct formal analysis of the model and to verify model properties specified in LTL. Each state diagram in the model is treated as a distinct Promela process, thus facilitating the interleaved execution that often reveals unexpected interactions among system components. The translation phase completes within two seconds for the Body Subsystem model.

6.5 Phase IV: Discovery of Latent System Properties (Marple)

In the fourth and final phase, the Promela model that was produced by Hydra is provided as input to Marple, which generates a suite of LTL properties that are presented to the developer in natural language. If a property is desirable, then it is added to the list of system requirements. A property that is undesirable must be addressed by the system's developer. Potential problems created by unwanted properties include incorrect functional behavior, feature interactions, distributed behavior problems, and behavioral inconsistencies. This phase takes on the order of six hours to complete on a 1.8 GHz PC with 16 GB of memory. According to our industrial collaborators, this time frame was well within the acceptable range given the potential severity of errors found. In our experiments, Marple was configured to return 25 properties.

Next, we present a sample set of latent properties that were discovered in the Body Subsystem. We provide a natural language representation of each property along with a brief discussion of the property, its consequences, and the mitigation strategy that was used.

Property 1:

`Globally, WiperModes.WiperMaster != RSM eventually holds`.

Property **1** states that the `WiperMaster` attribute in the `WiperModes` class must eventually have a value that is not `RSM` (Rain Sensor Mode). The developer determined that one of the state transitions in the `WiperModes` state diagram was missing a guard. Therefore, the transition was always available to be executed. Once the missing guard was added, we verified that the property no longer held.

Property 2:

`Globally, it is always the case that if DrvrDrSwitch.Switch == 1 holds, then Voltage_Range_Monitor.VBattRaw != 18 previously held`.

Property **2** states that if the `Switch` attribute in the `DrvrDrSwitch` (Driver Door Switch) class has a value of 1 then the value `VBattRaw` attribute of the `Voltage_Range_Monitor` class must not have been 18 in the previous state. Once the property was identified, the model developer was able to identify a missing assignment statement (`battStatus = NORM`) for the `INITIAL` state in the `VoltageRangeMonitor` state diagram. After the missing assignment statement was added, the property no longer held.

Property 3:

`Globally, it is always the case that if WiperModes.Command == 5 holds, then AmbientLightSensorInput.lightLevel != 4 previously held.`

Property **3** states that if the value of the `Command` attribute in the `WiperModes` class is `HALT`, then the value of the `lightLevel` attribute in the `AmbientLightSensorInput` class must not have been `TWILIGHT` in the previous state. Despite the different set of classes and attributes in this property as opposed to property **2**, the model developer discovered that Property **2** and Property **3** held because of the same missing assignment statement in the `VoltageRangeMonitor` state diagram. After the statement was added to remedy Property **2**, Property **3** no longer held.

Property 4:

`Globally, it is always the case that WiperModes.Command != HALT.`

Property **4** states that the value of the `Command` attribute in the `WiperModes` class will never be `HALT`. From this property, the model developer determined that a triggering event in the `RelayControl` class (part of the Windshield Wipers subsystem) never occurs, and thus the state machine remains in the `WAIT` state indefinitely. Figure 2 shows partial state diagrams from the `RelayControl` and `WiperModes` classes. There was a missing call to the event `RlyCtlActive` (shown in **bold**) in the transition expression for the initial state in `RelayControl` (Figure 2(a)). Since the transition expression for `WiperModes` (Figure 2(b)) is waiting for the event to be fired (also shown in bold), it will wait indefinitely. After adding a call to the missing event in the appropriate state transition in `RelayControl`, the property no longer held.

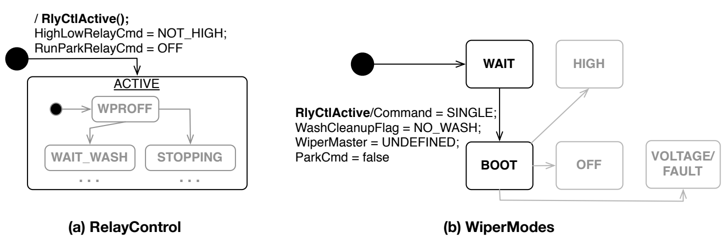

Fig. 2. Partial State Diagrams for Classes Affected by Property 4

7 Discussion

In this section, we present a discussion of the results of applying the proposed toolchain and consider the consequences of its use in an industrial development setting. As in previous sections, we present the discussion in terms of each phase of the toolchain.

7.1 Syntax and Semantics Defects

We had access to two major revisions of the Body Subsystem model for this work: an early revision that had not been used to generate source code and thus contained syntax errors and type inconsistencies, and a subsequent revision that had undergone source code generation and compilation. In order to assess Cyclops's ability to detect syntax and semantics errors, we applied it to the earlier model revision. Cyclops detected all of the errors that the compilers had detected during source code generation and compilation, and it also identified additional errors that were subtle and would be difficult to locate by manual inspection. For example, an assignment statement that was mistyped as a boolean comparison would not be detected by a compiler, but such a mistake may have an adverse effect on system behavior. The developer of the Body Subsystem model stated that without the use of a tool such as Cyclops, these subtle errors would have been allowed to propagate into generated source code and, perhaps, into the design and implementation of the system. Since system models are typically small during the late requirements stage of the software lifecycle, such defects are straightforward to resolve once they have been identified. Identifying and resolving these subtle defects in the requirements stage reduces the amount of time spent debugging and reengineering the system at later stages of development.

7.2 Latent Property Detection

While the proposed toolchain detects several types of model errors, the developer of the Body Subsystem told us that the toolchain is most useful for identifying portions of the model or system requirements that are missing. The toolchain identified a set of missing constant initializations, transition guards, and transition action statements. The discovered properties did not always point directly to the missing model components (e.g., properties 2 and 3 in Section 6), but they yielded enough information for a developer with knowledge of the system and model to make inferences about the possible causes of the defect and to revise the model accordingly. In the absence of the proposed toolchain, such defects would most likely be discovered during integration testing after the source code has been completed, thus increasing the cost to repair the defect.

Marple uses an evolutionary search technique to explore the space of properties for a given model. Due to inherent randomness in the search process, it is unlikely that Marple will revisit the same property in independent executions. However, it is straightforward to make note of any interesting properties and to re-examine them at a later time to monitor for regressions. The ability to track defects over time facilitated a step-wise, iterative model refinement process that enabled us to work with the model developer, who works with us remotely, to incrementally resolve the problems that our toolchain identified.

8 Conclusions

In this paper, we presented an experience report describing the use of a toolchain for detecting syntactic and semantics errors in behavioral system models, as well as detecting latent system properties during the early requirements phase of the software lifecycle. We demonstrated that the proposed toolchain is an effective means for identifying syntax errors, resolving ambiguous references, and discovering unwanted latent system properties.

We are considering several avenues for future work. First, we plan to integrate metamodel-level consistency checking into the Cyclops tool, thus enabling flexible and robust error detection that is grounded in a formal semantics for UML state diagrams. Next, we are investigating patterns within the discovered latent properties and to leverage their key features to fine-tune parameters for the Marple tool. Finally, we are exploring several strategies for reconfiguring the toolchain to detect situations in which two system features interact and lead to system failures or other unexpected behavior.

References

1. R.R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 126–133. IEEE, 1993.
2. R. Pressman. Software Engineering: A Practitioner's Approach. 2007.
3. Tigris.org. ArgoUML Modeling Tool. Available at http://argouml.tigris.org/, April 2011.
4. International Business Machines Corporation. Rational Software Architect. Available at http://www.ibm.com/developerworks/rational/products/rsa/, April 2011.
5. Microsoft Corporation. Available at http://office.microsoft.com/en-us/visio/, April 2011.
6. International Business Machines Corporation. Rational Rhapsody. Available at http://www-01.ibm.com/software/awdtools/rhapsody/, April 2011.
7. G.J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 2002.
8. G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.
9. P. Kars. The Application of PROMELA and SPIN in the BOS Project. In *The Spin Verification System: The Second Workshop on the SPIN Verification System: Proceedings of a DIMACS Workshop, August 5, 1996*, page 51. American Mathematical Society, 1997.
10. K. Havelund, M. Lowry, S.J. Park, C. Pecheur, J. Penix, W. Visser, J.L. White, et al. Formal analysis of the remote agent before and after flight. In *Lfm2000: Fifth NASA Langley Formal Methods Workshop*. Citeseer, 2000.
11. T. Bäck, D.B. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. Taylor & Francis, 1997.
12. J. Lehman and K.O. Stanley. Exploiting open-endedness to solve problems through the search for novelty. *Artificial Life*, 11:329, 2008.
13. W.E. McUmber and B.H.C. Cheng. A general framework for formalizing UML with formal languages. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 433–442. IEEE Computer Society, 2001.
14. G.J. Holzmann, American Telephone, and Telegraph Company. *Design and validation of computer protocols*, volume 94. Prentice Hall New Jersey, 1991.
15. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.
16. H. Goldsby and B. Cheng. Automatically Discovering Properties That Specify the Latent Behavior of UML Models. *Model Driven Engineering Languages and Systems*, pages 316–330, 2010.

17. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press, 1999.
18. Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proceedings of the International Conference on Software Engineering (ICSE05)*, St Louis, MO, USA, May 2005.
19. A. Egyed. Fixing inconsistencies in UML design models. In *Proceedings of the 29th international conference on Software Engineering*, pages 292–301. IEEE Computer Society, 2007.
20. A. Egyed. Scalable consistency checking between diagrams-The VIEWINTEGRA Approach. In *ase*, page 387. Published by the IEEE Computer Society, 2001.
21. G. Engels, J.M. Küster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. 26(5):186–195, 2001.
22. H. Gomaa and D. Wijesekera. Consistency in multiple-view UML models: a case study. In *Workshop on" Consistency Problems in UML-based Software Development II*, page 1. Citeseer, 2003.
23. C. Schwarzl and B. Peischl. Static-and dynamic consistency analysis of UML state chart models. *Model Driven Engineering Languages and Systems*, pages 151–165, 2010.
24. J. Simmonds, R. Van Der Straeten, V. Jonckers, and T. Mens. Maintaining consistency between uml models using description logic. *Série Lobjet-logiciel, base de données, réseaux*, 10(2-3):231–244, 2004.
25. R. Wagner, H. Giese, and U. Nickel. A plug-in for flexible and incremental consistency management. In *Proc. of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-based Software Development), San Francisco, USA*, 2003.
26. F. Baader. *The description logic handbook: theory, implementation, and applications*. Cambridge Univ Pr, 2003.
27. William Chan. Temporal-logic queries. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 450–463, London, UK, 2000. Springer-Verlag.
28. Richard M. Chang, George S. Avrunin, and Lori A. Clarke. Property inference from program executions. Technical Report UM-CS-2006-26, University of Massachusetts, 2006.
29. Arie Gurfinkel, Marsha Chechik, and Benet Devereux. Temporal logic query checking: A tool for model exploration. *IEEE Transactions on Software Engineering*, 29(10):898–914, 2003.
30. Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476, 2005.
31. Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM.
32. Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Propel: an approach supporting property elucidation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 11–21, New York, NY, USA, 2002. ACM.