Automatically Discovering Properties That Specify the Latent Behavior of UML Models*,**

Heather J. Goldsby and Betty H.C. Cheng

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

Abstract. Formal analysis can be used to verify that a model of the system adheres to its requirements. As such, traditional formal analysis focuses on whether known (desired) system properties are satisfied. In contrast, this paper proposes an automated approach to generating temporal logic properties that specify the *latent* behavior of existing UML models; these are unknown properties exhibited by the system that may or may not be desirable. A key component of our approach is MARPLE, a evolutionary-computation tool that leverages natural selection to discover a set of properties that cover different regions of the model state space. The MARPLE-discovered properties can be used to refine the models to either remove unwanted behavior or to explicitly document a desirable property as required system behavior. We use MARPLE to discover unwanted latent behavior in two applications: an autonomous robot navigation system and an automotive door locking control system obtained from one of our industrial collaborators.

1 Introduction

One approach to ensuring that models used for model-driven development provide the desired behavior is to analyze them for adherence to system requirements [1,2,3]. This analysis, however, does not detect errors in *latent behavior*, the unspecified and potentially unwanted behavior of the model; these errors could then be propagated to the implementation and even deployed. Uchitel et. al have proposed an approach for detecting one form of latent behavior called implied scenarios as part of the process of synthesizing a model from scenarios [4]. However, preexisting UML models cannot make use of this technique. Three broad categories of approaches have been developed to produce properties that could be used for analysis: *Requirements discovery approaches* (e.g., [5]) examine testing

* This work has been supported in part by NSF grants EIA-0000433, CNS-0551622, CCF-0541131, IIP-0700329, CCF-0750787, Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, Siemens Corporate Research, and a Quality Fund Program grant from Michigan State University.

** We gratefully acknowledge the feedback and insight provided by the reviewers of our earlier work.

D.C. Petriu, N. Rouquette, Ø. Haugen (Eds.): MODELS 2010, Part I, LNCS 6394, pp. 316–330, 2010.

and deployment artifacts to detect missing or erroneous properties; process improvements have been proposed as part of these approaches. *Refinement-based approaches* (e.g., [6]) infer properties from formally specified goals or requirements. Lastly, *specification generation techniques* (e.g., [7,8,9,10,11,12,13,14,15]) infer properties from a representation of a system (e.g., a model or code) or a derivative of the system (e.g., execution traces). Several previously developed specification generation approaches are able to infer temporal logic properties from a model [8,12], code [14], or execution traces [9,15]. For these approaches, the developer identifies a part of the system behavior to explore, either by restricting the exploration to a portion of the code [14], or by explicitly selecting the states, events, and variables that are of interest [8,9,12,15]. One ramification of having the developer guide the exploration is that the unexplored portions of the system may still conceal latent unwanted behavior. Ideally, developers would like to maximize both automation of property discovery and coverage, while minimizing the number of properties that must be examined.

In this paper, we propose an evolutionary-computation approach called MARPLE[1] to automatically generating properties that specify the latent behavior of UML models comprising an instance (class) diagram and multiple state diagrams. Evolutionary computation methods, such as genetic algorithms and genetic programming, have achieved considerable success, in some cases producing human-competitive designs [16]. Each evolutionary algorithm experiment comprises a population of individuals. Over the course of many generations, where each generation is subjected to natural selection, mutation, and crossover, the evolutionary algorithm seeks to optimize according to a *fitness function* that describes one or more objectives. For this approach, we use a recently developed technique called *novelty search*, where the objective is not to find one optimal solution, but rather to find a suite of sufficiently different solutions [17]. We use novelty search to enable MARPLE to produce properties that maximize coverage of the model's behavior, while minimizing human effort.

MARPLE uses novelty search to discover a set of properties that describe a UML model, where these properties describe behavior not explicitly stated in the requirements and may, in fact, be unacceptable latent behavior. Specifically, each individual within MARPLE represents a property created by instantiating one of the five most commonly occurring specification patterns [18] in the form of Linear Temporal Logic (LTL). Instantiating a pattern involves replacing the placeholders with evolved boolean propositions, where a proposition is created using attribute and operation information from a UML instance diagram of the system. Because the propositions can include conjunctives and disjunctives, the set of possible propositions is unlimited and too large for brute force search methods to explore. During the evolutionary process, mutations and crossover produce different LTL properties that may be satisfied by the UML model. The novelty of a property is assessed using the Spin model checker [19]. Specifically, the state space of the shortest *witness trace* (i.e., path that supports the property)

[1] MARPLE is named after Miss Marple, Agatha Christie's detective who was famous for detecting latent human behavior.

through the Spin representation of the model is compared to the state spaces of other properties. [2] If a *novel region* of the model state space is discovered (i.e., a region of the state space that has not been explored by previously evaluated properties), then the property is assigned a higher fitness value and MARPLE searches the new region more thoroughly. However, if a property explores a previously explored region of the state space, then it is assigned a low fitness value and MARPLE does not search the region as thoroughly. In this way, MARPLE discovers properties that cumulatively describe the behavior of the model. For readability purposes, the properties generated by MARPLE are presented to the developer in natural language [20] for assessment. The generated properties can be used by the developer to refine the requirements specifications (to explicitly sanction the latent behavior) or to modify the UML model (to remove unwanted latent behavior).

Overall, our approach enables developers to automatically explore UML models for properties representing potentially unwanted latent behavior. We illustrate our approach by using MARPLE to discover the unwanted latent behavior of models for an automobile door locking system obtained from one of our industrial collaborators. To further validate our approach, we have also applied it to a robot navigation system [21] and sought feedback from our industrial collaborators. The remainder of the paper is organized as follows. Section 2 presents relevant background information. Section 3 describes MARPLE. and describes results from our door locking case study. Section 4 describes how we validated the performance of MARPLE. Section 5 discusses related work. Finally, in Section 6, we present conclusions and discuss future work.

2 Background

In this section, we provide background information on the property specification patterns used for this approach, genetic programming, and novelty search.

2.1 Property Specification Patterns

Dwyer *et al.* identified several property specification patterns [18] that are commonly used to analyze systems for assurance needs. For our approach, we enable MARPLE to instantiate the five most common patterns (*Absence*, *Universality*, *Existence*, *Precedence*, and *Response*), using the global scope of applicability. [3] Additionally, to facilitate assessment by human developers, we use a previously developed structured English grammar [20] for the specification patterns to present relevant properties in natural language. Figure 1 depicts the natural language representations of these patterns, where p and q are placeholders for propositional expressions.

[2] Spin provides configuration options for generating the shortest path, which is how we produce the shortest witness trace.

[3] MARPLE can also be used to generate properties using other scopes. However, for brevity, we only present the global scope.

Pattern Name	Natural Language
Absence	Globally, it is never the case that p holds.
Existence	Globally, p eventually holds.
Universality	Globally, it is always the case that p holds.
Precedence	Globally, it is always the case that if p holds, then q previously held.
Response	Globally, it is always the case that if p holds, then q eventually holds.

Fig. 1. Global Specification Patterns

2.2 Genetic Programming

Genetic programming is an evolution-inspired approach to discovering computer programs that solve a problem. A genetic programming experiment comprises a *population* of individuals, where each individual is a program tree. Figure 2(a) depicts one such program tree that represents the function: x + (sin(x) * 5). Each node in the tree can be a *function* that takes one or more parameters that are represented as subtrees (e.g., +, *, sin), or a *terminal* that does not take any parameters and may represent either a variable or a constant (e.g., x, 5). At the start of the experiment, a population of individuals is created using a random assortment of the available functions and terminals. Each individual has an associated fitness that represents how closely it approximates the solution, e.g., the desired symbolic regression formula (x*x*x).

A genetic program consists of many generations of individuals. During each generation, the fitness of the individuals is evaluated. Then individuals with high fitness scores are selected to be used to create the subsequent generation. An individual may be selected for *mutation*, where one or more nodes within the program tree are changed to another node prior to being placed within the next generation. For example, Figure 2(b) depicts how the tree in (a) could have been mutated by replacing the multiply function (*) with the divide function (/) effectively changing the formula to be: x + (sin(x) / 5); shading denotes the point of change. An individual could also be selected for *crossover*, where a subtree is exchanged with a subtree from another selected individuals. For example, Figure 2(c) depicts how the tree in (a) could have been modified by crossover replacing the sin(x) subtree with the constant 7. Because the highly fit individuals are preferentially selected to be used as the raw material to create subsequent generations, over time, the solutions discovered by the genetic programming experiment optimize according to the fitness function.

In this paper, we use genetic programming as the basis for our representation of LTL properties, where each program tree represents one property and the terminals represent concepts defined by the UML model.

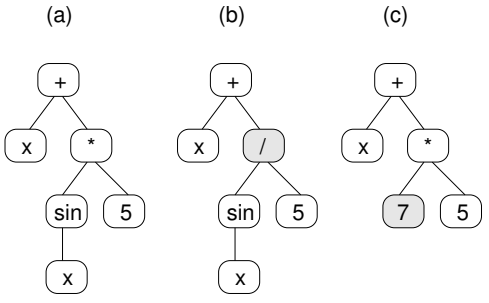

Fig. 2. Examples of a genetic program tree, where (a) is a tree, (b) is the same tree after a mutation, and (c) is the same tree after crossover. The shaded nodes represent the location of the change.

2.3 Evolutionary Computation and Novelty Search

In general, evolutionary computation is a search technique used to explore large and complex search spaces for solutions that optimize a fitness function. However, the application of a fitness function can sometimes be shortsighted leading to evolutionary computation approaches becoming "stuck" on sub-optimal solutions that represent local minima, rather than discovering the more complex and better solution. Lehman and Stanley originally developed novelty search, where fitness is a measure of how rare the behavior of an individual is, as a method for discovering more complex and better solutions [17]. Specifically, novelty search uses the following fitness formula:

$$\rho(x) = \frac{1}{k} \sum_{i=0}^{k} dist(x, u_i)$$

where $\rho(x)$ is the novelty measurement for individual x; k is the number of nearest neighbors used for the novelty calculation; and $dist(x, u_i)$ is the distance between individual x and its i^{th} nearest neighbor, u_i. To calculate $\rho(x)$ the distance between x and all other individuals in the current population and the *archive* of previously discovered novel individuals is computed. The novelty metric is then computed by taking the mean of the distance to the k nearest neighbors. If the novelty value was greater than ρ_{min}, then the individual is entered into the archive. In this way, individuals that explored previously unseen areas of the search space were assigned a higher fitness. This technique has produced a neural net that enabled a robot to more effectively navigate the maze, as compared to neural nets created using evolutionary computation techniques that sought to maximize fitness, rather than novelty [17].

While in previous work novelty search was used to discover better solutions than other evolutionary computation techniques, in this paper, we use novelty search to discover a suite of properties that cumulatively attempt to cover the state space of a model.

3 Approach

At a high level, our approach uses novelty search to mine a model for properties that may represent either known sanctioned behavior or unknown latent behavior. Three steps are used for running MARPLE:

1. The developer configures MARPLE for a specific model.
2. The developer runs MARPLE to produce a set of properties.
3. The developer reviews the properties and uses the information to improve the model.

In this section, we provide further detail about this process using a door locking model obtained from industry as a running example.

3.1 Case Study

We illustrate our approach by applying it to an automobile door-locking system that was obtained from our industrial collaborators. Figure 3 depicts an object diagram for the system. The door-locking system is a distributed embedded system responsible for controlling the centralized door locks in a car. The door-locking system comprises two control units, placed in the driver and passenger doors, respectively. The units control the sensors and actuators located on the respective sides of the car. To lock and unlock the doors, the locks on the driver or passenger door may be used by inserting and turning a key in the key cylinder. All doors in the car will be locked or unlocked simultaneously. In addition, doors can be locked and unlocked from within the car, using a button located on each door. For safety reasons, unlocking always has priority over locking, so that in case of emergencies the car can be exited quickly.

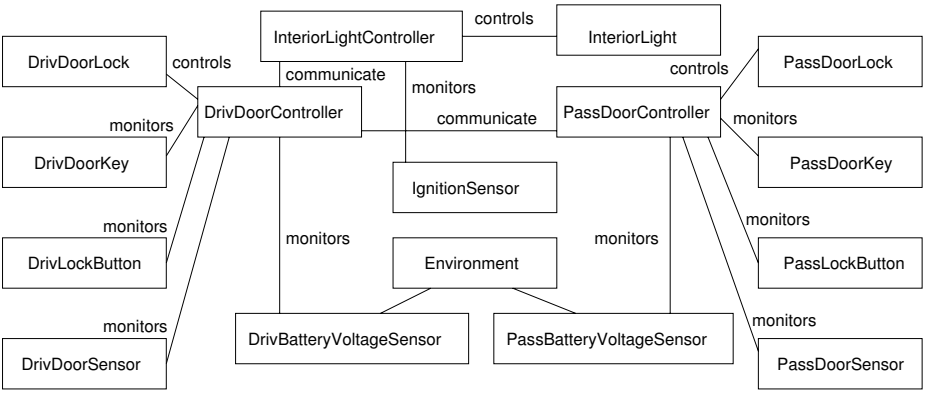

Fig. 3. Door Locking System Object Diagram

3.2 Step 1: Configuring Marple

To use Marple, a developer needs to provide: (1) a UML model that includes an object diagram and a set of state diagrams, where each state diagram describes the behavior of one object, and (2) a textual representation of the attributes and methods of the model. These attributes and methods are used as building blocks to create propositional expressions that replace the placeholders in the specification patterns. For example, Figure 4 depicts a snippet of the text file used to create the propositional expressions. For each operation, a propositional expression representing the operation being called is created. For each boolean attribute, a terminal where the attribute is true and another terminal where the attribute is false is created. For example, basic propositions DrivDoorController.doorStatus == 0 and DrivDoorController.doorStatus == 1 are among the propositions created for line 4. For each provided value of an integer, we create terminals where the attribute is equal to the value and when the attribute is not equal to the value. For example, basic propositions DrivDoorController.batteryVoltage == 6 and DrivDoorController. batteryVoltage != 6 are among the propositions created for line 2. The door locking system has 143 unique basic propositional expressions. Within Marple these basic expressions are then combined using conjunctives and disjunctives to form more complex propositional expressions.

```
1   classname DrivDoorController
2   attribute batteryVoltage int 6 9
3   attribute keyStatus int 0 1 2
4   attribute doorStatus boolean 0 1
5   attribute lockButtonStatus int 0 1 2
6   attribute iterations int 0 1 2 3 4
7   attribute initSuccess boolean 0 1
8   attribute voltageSuccess boolean 0 1
9   operation setBatteryVoltage
10  operation setKeyStatus
11  operation setDoorStatus
12  operation setLockButtonStatus
```

Fig. 4. An elided portion of the text file used to create model-specific terminals

3.3 Step 2: Marple

Given the inputs provided by the user as part of Step 1, Marple automatically produces a suite of LTL properties specified in natural language that cumulatively capture the requirements and latent behavior of the model. Here we describe: (1) how Marple internally represents properties, including how properties are mutated and crossed-over and (2) how the fitness function that governs the behavior of the evolutionary algorithm works.

Internal Property Representation. Essentially, each individual within a Marple experiment represents a property as a Genetic Program. When Marple starts, it randomly creates a population of these trees representing different properties.

Figure 5 depicts two such individuals and the natural-language representation of the property that they specify. To enable MARPLE to evolve such trees, we created a set of function nodes that are used to create properties for all possible models and a set of terminals that are propositional expressions and are model specific. Specifically, we provided function nodes for Absence, Existence, Universality, Precedence, and Response properties. Each tree had to be rooted with one of these nodes. Each of these nodes took a specific number of subtrees that correspond to the number of placeholders for propositional expressions. Next, we created two additional function nodes and and or, which are used to create more complex propositional expressions. For example, Figure 5(a) is an Absence property that contains a subtree with an or node.

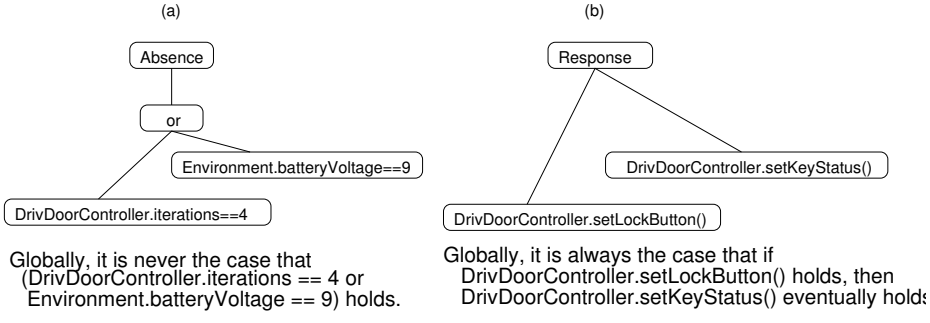

Fig. 5. Two properties generated by MARPLE. Property (a) represents unwanted latent behavior. Property (b) represents acceptable latent behavior.

If an individual property is selected for mutation, then one of its nodes or terminals is randomly exchanged with another node or terminal. Because MARPLE respects the type of a given node (e.g., a node representing an absence property will only be replaced with a node representing a different type of property), the produced property will always be syntactically correct. For example, after mutation, the property Figure 5 (a) may turn into property Globally, it is never the case that DrivDoorController.batteryVoltage == 9 holds., which changes the boolean expression from a disjunctive expression to a basic proposition. Another possible property that could be constructed by mutating the property depicted in Figure 5 (a) is Globally, it is always the case that DrivDoorController.iterations == 4 or Environment.batteryVoltage==9 holds, which changes the type of property being specified from an absence property to a universality property.

If two properties are selected for crossover, then subtrees of the properties are exchanged. For example, if properties (a) and (b) in Figure 5 are selected for crossover, then the resulting properties might be:

- Globally, it is never the case that DrivDoorController.setLockButton() holds.
- Globally, it is always the case that if (DrivDoorController.iterations == 4 or Environment.batteryVoltage == 9) holds, then DrivDoorController.setKeyStatus() eventually holds.

where the underlined portions of the properties represent the parts that have been exchanged through crossover.

Fitness Function. The central aspect of measuring the novelty of a property is the distance metric that measures how similar (or different two properties) are. In this case, we use the novelty search function described in Section 2.3 and define distance as the difference between the state spaces covered by the *witness traces* of the respective properties, where a witness trace is the shortest path of execution through a model that satisfies a property. Specifically, first, we use the Spin model checker [19] to verify the property holds. Then, if it does, we invert the property to produce a witness property. For example, the witness property of the property described in Figure 5 is: *Globally, it is eventually the case that (DrivDoorController.iterations ==4 or Environment.batteryVoltage == 9) holds.* Execution paths that violate the witness property are execution that satisfy the original property.

This distance measurement has several associated benefits. First, if the property does not hold for the model, then the set of states is empty. If the property is trivially true (e.g., the situation where the proposition x is always false and thus the property Globally, it is always the case that if x holds, then y eventually holds. is vacuously true), then the set of states is also empty. In this way, the distance metric compresses uninteresting properties together and enables us to discover more novel properties that explore different areas of the state space.

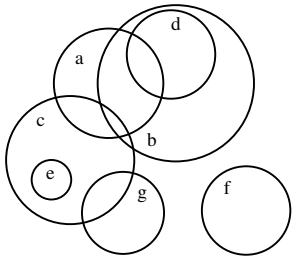

Fig. 6. Visualization of novelty metric

We then perform novelty search using the distance metric in order to discover latent model behavior. Figure 6 provides a graphical depiction of how novelty search works. For this example, we are assessing the novelty of property a. Each circle represents the state space of a property. If two circles overlap, then they share a common set of states (e.g., c and e, a and c). The distance between two circles is the set difference between their states. To compute the novelty of a property, we examine all possible pairs of properties both in the property and also in the archive of all previously generated properties. In this case, if k were equal to 2, then the nearest neighbors of property d would be properties b and a. The novelty of property d is the mean of the difference between d and b and the difference between d and a. Because we are interested in a suite of properties that cumulatively describe the behavior of the model, rather than a single penultimate property, each generated property is added to the archive.

3.4 Step 3: Assessing the Properties

At the end of a run, MARPLE returns the contents of the archive, which represents all of the generated properties, to the developer. These properties are provided in natural language, using Spider, a previously developed tool [20],

to facilitate understanding. The properties may represent either requirements, latent acceptable behavior, or latent unacceptable behavior.

To assist the developer in analyzing the properties, we use a two-step approach to using the properties to uncover latent behavior:

1. Inspect and assess the absence, universality, and existence properties generated by MARPLE. These types of properties are able to detect subtle errors, such as whether or not an attribute ever changed values or a method was called. Within this step, we begin by assessing the properties with novelty values greater than zero, since these represent the minimal set of discovered properties that effectively explore the state space and as such provide an overview of the behavior of the model. If a property specifies potentially unwanted behavior, we then *zoom in* by looking at other properties that use the same attribute and class names, as these properties may specify the same behavior in a manner more intuitive to understand.
2. Inspect and assess the precedence and response properties generated by MARPLE. These properties are able to detect timing errors or unwanted relationships among model elements. We repeat the magnification process by first focusing on the novel properties and then zooming in on behaviors of interest.

To illustrate this approach, we use the results from one run of MARPLE, which produced 351 properties, 167 of which were verified as describing the model, of which 63 had a novelty value greater than zero.

Figure 7 depicts five absence and existence properties generated by MARPLE for the door-locking model. By visually inspecting these properties, we were able to classify properties 2 and 5 as representing acceptable model behavior. However, properties 1, 3, and 4 represent unwanted latent behavior – all of these properties specify that certain attribute values were never used.

1. **Absence Property:** Globally, it is never the case that (Environment.batteryVoltage==9 or DrivDoorController.iterations==4) holds.
2. **Existence Property:** Globally, (DrivDoorController.batteryVoltage != 9 or DrivDoorController.initSuccess==1) eventually holds .
3. **Absence Property:** Globally, it is never the case that DrivDoorController.initSuccess==1 holds.
4. **Absence Property:** Globally, it is never the case that DrivBatteryVoltageSensor.voltage==9 holds.
5. **Existence Property:** Globally, PassDoorController.setKeyStatus() eventually holds.

Fig. 7. Five absence and existence properties generated by MARPLE

Next, we visually inspected the model in an attempt to discover the source of the unwanted behavior. While examining the state diagram for the DrivDoorController we noted a subtle, but important error: The DrivDoorController was

missing a transition that connected the start state to the initialization state Figure 8 depicts an elided version of the state diagram for the DrivDoorController. Specifically, to illustrate the error, the elided diagram depicts only the initialization state and the compound states. The bolded transitions were included in the model. The dotted line transition represents the missing transition. Because the DrivDoorController was missing this transition, it did not initialize properly and thus did not initialize other components. The PassDoorController also contained this error. We consider this error to be subtle and difficult to detect, given it strongly influenced the behavior of the controllers and yet this model had been developed and analyzed by a member of our lab for adherence to requirements.

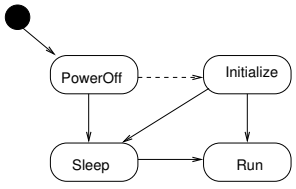

Fig. 8. The elided version of the DrivDoorController state diagram. The dotted line transition represents a missing transition that caused unwanted latent behavior.

Given serious unwanted latent behavior was detected early in the process, the model was corrected and the analyses rerun. However, if the absence, universality, and existence properties were acceptable, then we would expand our analysis to include the precedence and response properties.

4 Validation

While we have described the process of using MARPLE and have provided evidence that in one case MARPLE was able to detect unwanted latent behavior, To further validate our approach, we compare MARPLE's ability to discover properties that describe the door locking system to a control experiment that did not use novelty search.

The control version of MARPLE did not use novelty information for the evolutionary search process – properties were randomly selected for mutation, crossover, and survival. To ensure the generality of the control experiments, in addition to assessing MARPLE's performance on the door locking system, we also evaluated it on a model of an autonomous robot navigation case study, originally developed by Park et al. [21], and later revised and modeled by us [22]. To account for the stochastic nature of the evolutionary process, for each model, we ran 30 control runs and 30 MARPLE runs.

We then examined the central question of whether MARPLE is better able to identify novel properties than the control. Table 1 depicts the results. The first row is the total number of generated candidate properties, where these properties may have been either true or false for the model. For both systems, MARPLE generated over 35% more candidate properties than the control. The second row is the number of properties that were verified to describe the behavior of the model. Here, MARPLE produced more than twice as many properties that described the behavior of the model than the control. On average, over 45% of the properties generated by MARPLE were verified as describing the model. In

contrast, less than 30% of the properties generated by the control were verified. The last row is the number of properties whose witness traces included at least one previously unvisited state. These properties represent 30-40% of the verified properties generated by MARPLE and indicate that MARPLE is exploring novel regions of the state space. Overall, these results demonstrate that MARPLE is an effective strategy for exploring the behavior of a model. In general, these runs took approximately 8 hours. This time period enables a developer to run MARPLE overnight and have results in the morning.

Table 1. The mean number of properties generated by the control and MARPLE runs for the door-locking system and robot navigation system

	Door-Locking System		**Robot Navigation System**	
	Control	**Marple**	**Control**	**Marple**
Candidate Properties	214.78	294.20	205.31	287.14
Properties	59	134	56	141
Novel Properties	-	43.27	-	59.27

We solicited feedback from our industrial collaborators regarding the properties identified and the overall process. First, they confirmed that the time frame for generating properties (essentially overnight) and the number of properties generated was reasonable. Second, the format in which the generated properties were presented (i.e., structured natural language) was an effective format for developers to review and determine whether the properties represented sanctioned or unwanted behavior. Third, the assessment process that we proposed was useful and viable. In general, the feedback was that MARPLE provided a much needed means for detecting unwanted behavior in models of high assurance systems.

5 Related Work

In general, specification generation techniques produce properties by instantiating property patterns that contain *placeholders* with propositions that specify valid properties. Three major categories of related work are: static inference, dynamic inference, and temporal logic query checking. *Static inference* approaches infer properties from code specifications by analyzing program text (e.g., [7,14]) or by analyzing the code using a modular model checker [11]. *Dynamic inference* approaches infer likely properties, called invariants, from execution traces generated by code specifications (e.g., [9,10,15]). *Temporal logic query checking* (e.g., [8,12]) finds the strongest formulae adhered to by the model that satisfy the *temporal logic query*, which is a temporal logic formula with placeholders.

Next, we discuss the specification generation techniques that generate temporal logic properties [8,9,12,14,15]. Perracotta [15], a dynamic inference approach, generates eight variations of the temporal logic response pattern from imperfect

execution traces, where the developer instruments the program to monitor events and states of interest that constitute the possible propositions. Chang *et al.* [9] proposed a dynamic inference approach that generates temporal logic properties from a set of inference templates built using the Propel patterns [23]. Event traces are used to refine the inference templates to eliminate properties that are not satisfied by the program's event traces. Propositions based on developer-selected events are used to instantiate the property templates. Weimer and Necula proposed a static inference approach [14] to detecting bugs in source code. Their approach generates properties that specify the behavior of the error-handling code. Temporal logic query checking approaches [8,12] automatically find solutions to a temporal logic query. Specifically, to find the strongest formula, the query checker replaces the placeholders with combinations of developer-specified propositions.

In general, these approaches rely on developer knowledge (i.e., selected propositions to use and/or a selection of code) to determine the part of the system behavior to explore for properties. In essence, these approaches specify behavioral properties that refine the developer's understanding of a developer-specified segment of system behavior. Our approach can be used in a complementary fashion in that it identifies latent properties referring to propositions and/or properties *not explicitly* identified by the developer and that might otherwise remain concealed. For example, MARPLE could be used to identify unwanted latent behavior, and temporal logic query checking could be used to refine developer knowledge by identifying the strongest relationship among the MARPLE discovered propositions.

6 Conclusions and Future Work

In this paper, we have presented an approach to automatically generating properties that specify the unwanted behavior of UML models. Our approach relies on MARPLE, an evolutionary computation technique, to generate properties that are presented to the developer in natural language. Specifically, MARPLE generates properties by instantiating specification patterns with propositions developed using information in the UML class diagram. Regarding the scalability of this approach, as with most industrial uses of model checking and formal analysis, MARPLE is intended to be used on subsets of systems, particularly those of a critical nature. We have used MARPLE to detect latent properties of several models provided by our industrial partners in order to demonstrate its ability to work on models of industrial scale. Parallelizing the novelty search algorithm, changing the number of individuals within the population, or changing the number of generations that the algorithm runs could reduce the analysis time, and we will explore this optimization strategy in future work. Our approach is complementary to other specification generation techniques because it is able to identify unwanted latent behavior in portions of the model that may otherwise remain unexplored with the other approaches.

Our future work will explore extending Marple to use specification patterns that include additional scopes, as well as specification patterns for real-time

properties and other types of properties [20,23]. These extensions will enable Marple to detect additional sources of latent behavior. Lastly, we are investigating using MARPLE to detect feature interaction properties and automatically generate test cases for the corresponding code [24].

References

1. McUmber, W.E., Cheng, B.H.C.: A general framework for formalizing UML with formal languages. In: Proceedings of the IEEE International Conference on Software Engineering (ICSE 2001), Toronto, Canada (May 2001)
2. Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models. In: Proceedings of the 14th IEEE International Conference on Automated Software Engineering, Washington, DC, USA, p. 255. IEEE Computer Society, Los Alamitos (1999)
3. Tanuan, M.C.: Automated Analysis of Unifed Modeling Language (UML) Specifications. Master's thesis, University of Waterloo, Canada (2001)
4. Uchitel, S., Kramer, J., Magee, J.: Detecting implied scenarios in message sequence chart specifications. SIGSOFT Softw. Eng. Notes 26(5), 74–82 (2001)
5. Lutz, R.R., Mikulski, I.C.: Requirements discovery during the testing of safety-critical software. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering (2003)
6. Letier, E.: Reasoning about Agents in Goal-Oriented Requirements Engineering. PhD thesis, Louvain-la-Neuve, Belgium (2001)
7. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: ESEC-FSE 2007, pp. 25–34. ACM, New York (2007)
8. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000)
9. Chang, R.M., Avrunin, G.S., Clarke, L.A.: Property inference from program executions. Technical Report UM-CS-2006-26, University of Massachusetts (2006)
10. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27(2), 99–123 (2001)
11. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
12. Gurfinkel, A., Chechik, M., Devereux, B.: Temporal logic query checking: A tool for model exploration. IEEE Transactions on Software Engineering 29(10), 898–914 (2003)
13. Jeffords, R., Heitmeyer, C.: Automatic generation of state invariants from requirements specifications. SIGSOFT Softw. Eng. Notes 23(6), 56–69 (1998)
14. Weimer, W., Necula, G.C.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005)
15. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: ICSE 2006: Proceedings of the 28th International Conference on Software Engineering, pp. 282–291. ACM, New York (2006)
16. Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, M., Yu, J., Lanza, G.: Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Springer, Heidelberg (2003)

17. Lehman, J., Stanley, K.: Exploiting open-endedness to solve problems through the search for novelty. In: Bullock, S., Noble, J., Watson, R., Bedau, M.A. (eds.) Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems, pp. 329–336. MIT Press, Cambridge (2008)
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, pp. 411–420 (1999)
19. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2004)
20. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the International Conference on Software Engineering (ICSE 2005), St. Louis, MO, USA (May 2005)
21. Kim, M., Kim, S., Park, S., Choi, M.T., Kim, M., Gomaa, H.: UML-based service robot software development: a case study. In: ICSE 2006: Proceeding of the 28th International Conference on Software Engineering, pp. 534–543 (2006)
22. Goldsby, H.J., Cheng, B.H.C., McKinley, P.K., Knoester, D.B., Ofria, C.A.: Digital evolution of behavioral models for autonomic systems. In: Proceedings of the 5th International Conference on Autonomic Computing (ICAC 2008), Chicago, Illinois (June 2008)
23. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: Propel: an approach supporting property elucidation. In: ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pp. 11–21. ACM, New York (2002)
24. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: ROSATEA 2006: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, pp. 53–63. ACM, New York (2006)