An Approach to Analyzing Temporal Properties in UML Class Models

Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, and Indrakshi Ray
Colorado State University
Computer Science Department
{mustafa, rabdunab, france, iray}@cs.colostate.edu

Abstract. The Unified Modeling Language (UML) Class Models are widely used for modeling the static structure of object-oriented software systems. Temporal properties of such systems can expressed using TOCL, a temporal extension to the Object Constraint Language (OCL). Verification and validation of temporal properties expressed in TOCL is non-trivial and there are no automated tools that can aid such analysis. Existing approaches rely on transforming the UML models to another language that supports automated analysis. Such transformation is complex and can introduce errors. Towards this end, we propose an approach for directly analyzing temporal properties expressed in TOCL. We present a case study based on the Steam Boiler Control System to demonstrate the applicability of the approach.

Keywords: Analysis, Verification, Class Model, Temporal Properties

1 Introduction

The Unified Modeling Language (UML) Class Models are probably the most common specification diagrams used in the software industry. Automated analysis of class models often uncovers design problems. Detecting design problems in a timely manner saves time and effort. Specifying and analyzing temporal properties in class models are non-trivial. Consider the following temporal property in the Steam Boiler Control System [1]: "when the system is in the initialization mode, it remains in this mode until all physical units are ready or a failure of the water level measurement device has occurred." It is hard to express such property using Object Constraint Language (OCL) in a class model. TOCL [2], however, is a temporal logic extension of OCL and can specify such temporal properties. Once a property is specified, the class model must be analyzed to check for the satisfaction of such properties. To the best of our knowledge, we are unaware of any class model-based techniques for directly analyzing TOCL properties.

There are a number of model-checking based techniques for specifying and analyzing temporal properties in UML behavioral models, such as state machines and activity diagrams (e.g., see [3, 4]). These techniques involve developing an exogenous transformation, in which the source and target models are expressed

in different languages. Typically, the UML behavioral models are transformed to languages that are supported by model checking tools. There are three major challenges associated with these approaches: (1) effective use of these heavy-weight techniques requires developers to have specialized skills, (2) one has to prove that the transformations preserve the semantics of the source UML models, and (3) the results of the analysis performed by the back-end analysis tool must be presented to developers in UML terms, thus requiring another exogenous transformation.

However, temporal properties can also be expressed in class models that must be subsequently verified. One option is to transform them into other languages supporting automated analysis, as is done for the temporal properties specified on the behavioral models. But such an approach will have similar problems to those mentioned earlier. Another option is to develop model-checking support for verifying TOCL properties in UML class models with operation specifications. Given the complex state spaces that have to be codified and analyzed, this is a very challenging research problem.

Existing tools of UML/OCL such as USE [5] and OCLE [6] can be used to analyze structural properties, but they provide little support for temporal analysis. For example, the USE tool allows a user to interactively simulate the behavior of an operation by entering commands that change the states of objects and then the user checks if the operation's postconditions hold. Interactive simulation of operation behavior is useful, but can be tedious, time-consuming, and error-prone when manually simulating a scenario involving many interactions. Towards this end, researchers have demonstrated how scenarios can be modeled as a sequence of snapshots, which, in turn, can be verified using USE and OCLE [7]. However, adapting such an approach for verifying temporal properties is still an ongoing challenge. Our current work aims to fill this gap.

In this paper we propose a lightweight class model-based analysis approach that checks temporal properties against a non-exhaustive set of behavioral scenarios. The approach neither requires the use of exogenous transformations nor specialized skills other than those related to UML modeling. Our approach builds upon our previous work [8], where we described a temporal analysis approach that leverages the USE Model Generator [9] to produce a subset of the class model state space. A TOCL property is then checked against this state space. The approach described in this paper improves upon the earlier version in the following manners. First, the new version of the approach is based on the USE Model Validator which significantly outperforms the Model Generator [9]. The Model Validator uses boolean satisfiability (SAT) solvers to perform the analysis task. This results in a larger set of behavioral scenarios that can be checked and hence increases the confidence that a temporal property holds on a class model. Second, in the previous version, a procedure for creating non-exhaustive set of scenarios is defined manually. This task is tiresome, error-prone, and can be difficult to non-experts. In the current approach, this step is automated. Lastly, a complementary step is added to make the analysis results easier to exam to find the error. We applied our approach in specifying and analyzing real temporal

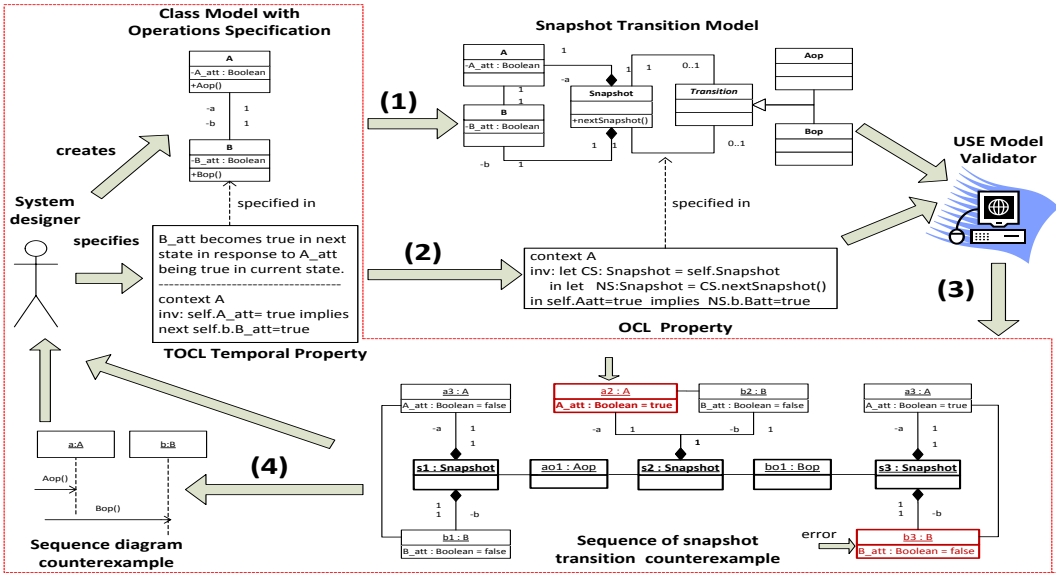

Fig. 1: An Overview of the Approach

properties of the Steam Boiler System. Note that, checking such properties is non-trivial using our earlier approach [8].

The rest of the paper is organized as follows. In Section 2, we give an overview of the proposed analysis approach. Section 3 presents the specification Steam Boiler System properties and Section 4 illustrates the analysis of these properties. In Section 5, we discuss related work, and in Section 6 we summarize our contributions and give pointers to future directions.

2 An Overview of the Approach

The research that led to this approach focused on answering the following question: "Given a UML class model, and a temporal property, is there a scenario supported by the class model that violates the property?." Figure 1 presents an overview of the approach. At the front-end of the approach, a system designer is responsible for 1) creating a design class model, and 2) specifying a temporal property in TOCL. A class model specifies application states and includes OCL specifications of operations. Then, the USE Model Validator is used at the back-end to generate behavioral scenarios against which the temporal property is checked. The tool produces a *scenario*, an object diagram of snapshot transition, that violates the temporal property. The back-end processing is transparent to the system designer.

The approach consists of four major steps. A transition-based class model of behavior is produced in Step 1. The model, called a *Snapshot Transition*

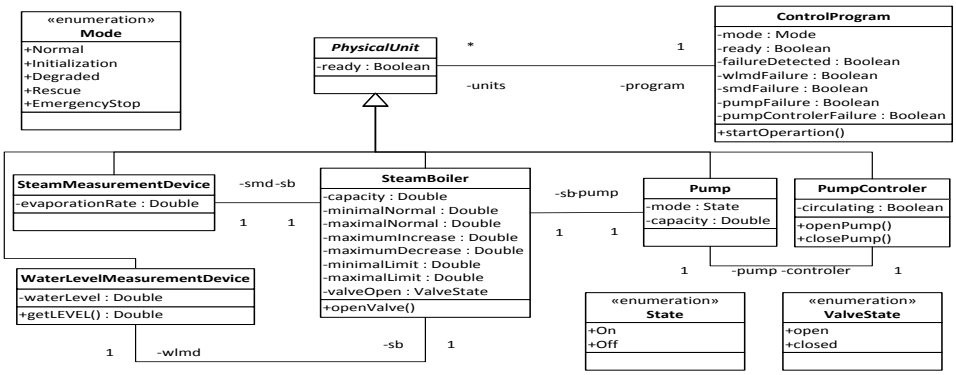

Fig. 2: The Design Class Model for the Steam Boiler Control System

Model (STM), is a class model that characterizes the valid sequences of state transitions caused by executions of operations specified in the class model. A state is modeled as a configuration of objects called a snapshot. The STM is mechanically generated from the class model.

In Step 2, the temporal property to be checked is converted to an OCL property defined in the context of the STM. The temporal property is specified in TOCL, a temporal logic extension to OCL [2]. The TOCL property and its OCL representation are instances of temporal property specification patterns that enable the UML modelers to apply reusable solutions to specify temporal properties in object-oriented notation, more details in our paper [8].

In Step 3, the USE Model Validator tool is used to produce instances of the STM (scenarios) and check the STM constraint generated in Step 2 against the scenarios. Specifically, the tool checks if there is a scenario that violates the temporal property.

The analysis results for scenarios that have a large number of snapshots and transitions might be difficult to interpret. For ease of examining, this result is also visualized by a UML sequence diagram in Step 4.

3 The Steam Boiler Control System Problem

We use the Steam Boiler Control System described in [1] to illustrate the proposed approach. The system works correctly when the water level is within two normal limits (minimalNormal and maximalNormal) and can not pass over two critical limits (minimalLimit and maximalLimit). Otherwise the steam-boiler can be seriously damaged.

Figure 2 shows a design class model of the Steam Boiler Control System.

The class model has five operations that change the state of the system. The operation getLEVEL() reads the water level and stores it in the variable waterLevel and getSTEAM() reads the evaporation steam rate and writes it in the

Table 1: TOCL and OCL specification of the steam boiler temporal properties

Temporal Property	Pattern	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
TP1: As soon as the program recognizes a failure of the water measuring device unit it goes into the rescue mode.	Response-global	**context** *ControlProgram* **inv:** *self.wlmdFailure implies next self.mode=# Rescue*	**context** *ControlProgram* **inv:** *inv: let CS: Snapshot= self.snp in NS: Snapshot= CS.getNext()* **in** *self.wlmdFailure implies NS.program.mode= # Rescue*
TP2: Failure of any physical units except the water measuring device puts the program into degraded mode	Response-global	**context** *ControlProgram* **inv:** *(smdFailure or pumpFailure or pumpControlerFailure) implies next self.mode=# Degraded*	**context** *ControlProgram* **inv:** *let CS: Snapshot= self.getCurrentSnapshot()* in let NS: Snapshot = CS.$getNext()$ **in** *(self.pumpControlerFailure or self.pumpFailure or self.smdFailure) implies NS.program.mode =# Degraded*
TP3: If the water level is risking to reach one of the limit values (e.g., greater than maximalNormal or less than minimalNormal) the program enters the mode emergency stop.	Response-global	**context** *SteamBoiler* **inv:** *(self.wlmd.waterLevel > self.maximalNormal or self.wlmd.waterLevel < self.minimalNormal) implies next self.program.mode = # EmergencyStop*	**context** *SteamBoiler* **inv:** *let CS: Snapshot = self.snp* in let NS: Snapshot = CS.$getNext()$ **in** *(self.wlmd.waterLevel > self.maximalNormal or self.wlmd.waterLevel < self.minimalNormal) implies NS.program.mode = # EmergencyStop*
TP4: when the valve of the steam boiler is open, then eventually the water level will be lower or equal to the maximal normal level.	Response-global	**context** *SteamBoiler* **inv:** *self.valveOpen = # open implies sometime (self.wlmd.waterLevel < = maximalNormal)*	**context** *SteamBoiler* **inv:** *let CS: Snapshot = self.snp* in let FS: Set(Snapshot) = CS.$getPost()$ **in** *self.valveOpen = # open implies FS → exists (s:Snapshot | s.WLMD.waterLevel < = maximalNormal)*
TP5: when the program is in the initialization mode and a failure of the water level measurement device is detected it puts the program in the emergency stop mode.	Response-global	**context** *ControlProgram* **inv:** *(self.mode = # Initialization and self.wlmdFailure) implies next self.mode =# EmergencyStop*	**context** *ControlProgram* **inv:** *let CS: Snapshot = self.snp in let NS: Set(Snapshot) = CS.getNext()* **in** *(self.mode = # Initialization and self.wlmdFailure) implies NS.program.mode = # EmergencyStop*
TP6: when the system is in initialization mode, it remains in this mode until all physical unites are ready or a failure of the water level measurement device has occurred.	Universality-between Q and R	**context** *ControlProgram* **inv:** *self.mode = # Initialization implies always self.mode = # Initialization until (PhysicalUnit.allInstances→ forAll(u: PhysicalUnit | u.ready))*	**context** *ControlProgram* **inv:** *let CS: Snapshot = self.snp in let FS_1: Snapshot = CS.getPost() → select(s:Snapshot | s.boiler.ready and s.SMD.ready and s.pump.ready and s.PC.ready and s.WLMD.ready)→ first() in let $PreFS_1$=Set(Snapshot) = FS_1.getPre() in let BTS: Set(Snapshot)=$PreFS_1$ → excluding(CS.getPre())* **in** *self.mode = # Initialization implies BTS → forAll (s1:Snapshot | s1.program.mode= # Initialization)*

vaporationRate variable. The openPump(),closePump(), and openValve() operations open the pump, close the pump, and open the boiler valve, respectively. The OCL specifications of getLEVEL() and openPump() are defined bellow.

```
context WaterLevelMeasurementDevice::getLEVEL(): Double
pre: self.program.mode= #Normal
post: self.waterLevel = result

context PumpControler::openPump()
pre: self.pump.mode = # Off
post: self.pump.mode = # On
```

The system has a number of temporal requirements that need to be verified. We resorted to the use TOCL to specify the temporal properties in the boiler system. Table 1 presents the TOCL specifications of some of these properties. In our previous paper [8], we explain how to use reusable solution patterns to specify temporal properties in TOCL.

4 Case Study: Specifying and Analysing Temporal Properties of Steam Boiler

The following discusses the steps in Figure 1 in the context of the Steam Boiler Control System.

4.1 Step1: Generation of the *STM*

Step 1 takes the steam boiler class model (see Fig. 2) as input and produces a STM model [7]. The STM model characterizes a sequence of state transitions of the boiler system, where each transition is triggered by an operation invocation. The STM is formed by (1) creating a Snapshot class, (2) creating a hierarchy of transition classes representing operation invocation, and (3) converting operation specifications to invariants of the transition classes. Everything else (class invariants, associations ect.) remains intact in the STM model. Figure 3 shows the STM model that is produced from the boiler class model.

Each instance of the Snapshot class represents a state in a transition system. A snapshot is a configuration of one object of each of the concrete classes inf Figure 2. In this system, a snapshot has only one object of each class. The approach can also be used to specify snapshots that have many objects of each class, and it distinquishes between these objects using identifiers [7].

To create the hierarchy of transition classes, we generate a subclass of the abstract $Transition$ class from each operation. In the Steam Boiler class model, we only consider five modifier operations and thereby we create five subclasses of the class $Transition$, one for each operation (see Fig.2 and Fig. 3). For each parameter of an operation, we generate two references (shown as attributes of the Transition subclasses) that represent the value of the parameter before and after the execution of the operation. In boiler class model, none of the operations has a parameter, so we do not create any references. We define two references for each operation that point to the object's states before and after an operation invocation. A reference is also created for the return value of an operation.

We define the before and after state conditions in the STM as invariants based on the pre and post conditions of the operations in the initial class model.

The following illustrates how the getLevel() operation in the WaterLevelMeasurmentDevice class is defined in the STM model. We generate two references (wlmdPre and wlmdPost of type WaterLevelMeasurmentDevice) that point to the object that the operation is invoked on. Because this operation has a return value of type Double, a ret:Double reference is created to point to the returned value of the operation. The pre and post conditions of getLevel() operation (presented above) are converted to invariants in STM as follows:

```
context WaterLevelMeasurmentDevice_getLevel
inv: self.wlmdPre.program.mode=# Normal
inv: wlmdPost.waterLevel= ret
```

Similarly, we generate invariants from the pre and post conditions for all the other operations.

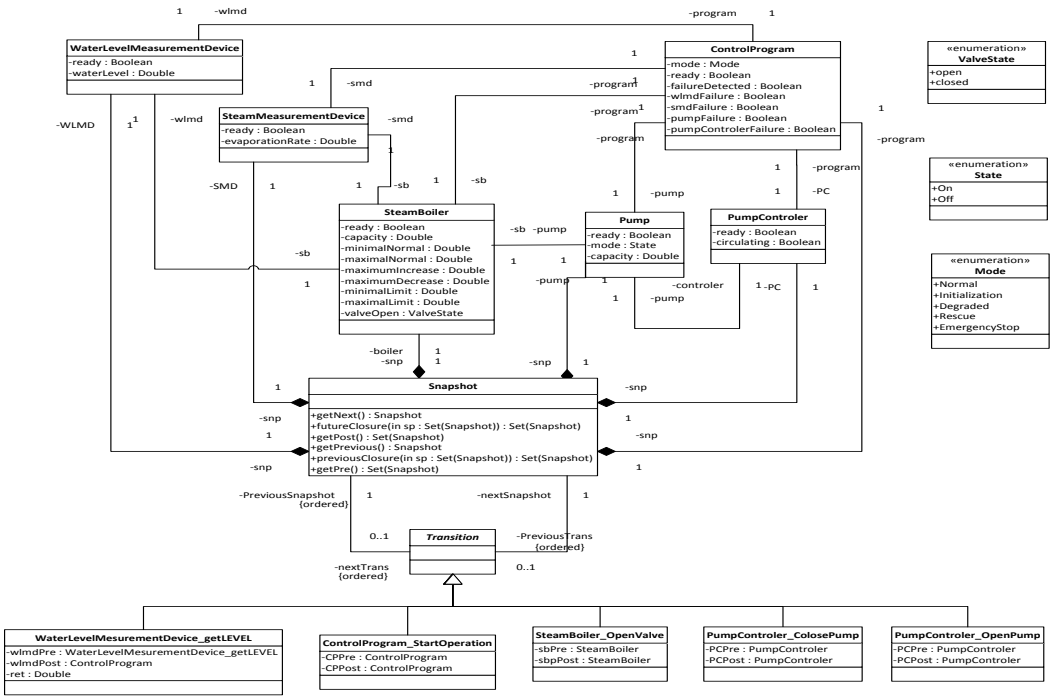

Fig. 3: The Steam Boiler Snapshot Transition Model

4.2 Step2: Converting TOCL to OCL properties

In this step, the steam boiler class model is unfolded as a sequence of snapshot transitions represented by the STM in order to express TOCL properties as OCL constraints. We first specify the temporal properties in the Steam Boiler Control System using TOCL. Table 1 shows some of the boiler system TOCL properties. These TOCL properties are specified in the context of the steam boiler class model. Then, OCL constraints are systematically produced in the context of the steam boiler STM model (see Fig. 3) using these TOCL properties. Each OCL constraint captures the semantics of the corresponding TOCL constraint in the context of the STM model.

Consider the TOCL and OCL expressions of the temporal property TP1 in the Table 1. The TOCL states that if the water measuring device fails *(self.wlmd Failure=true)* then the program goes into the rescue mode ($nextself.mode = \#Rescue$). In the corresponding OCL expression, the next state (NS) is returned by first getting the current *snapshot(CS)* and navigating to the next state by the operation *getNext()*. Then the OCL asserts that if the water measuring device fails, then the program in the next state is in the Rescue mode.

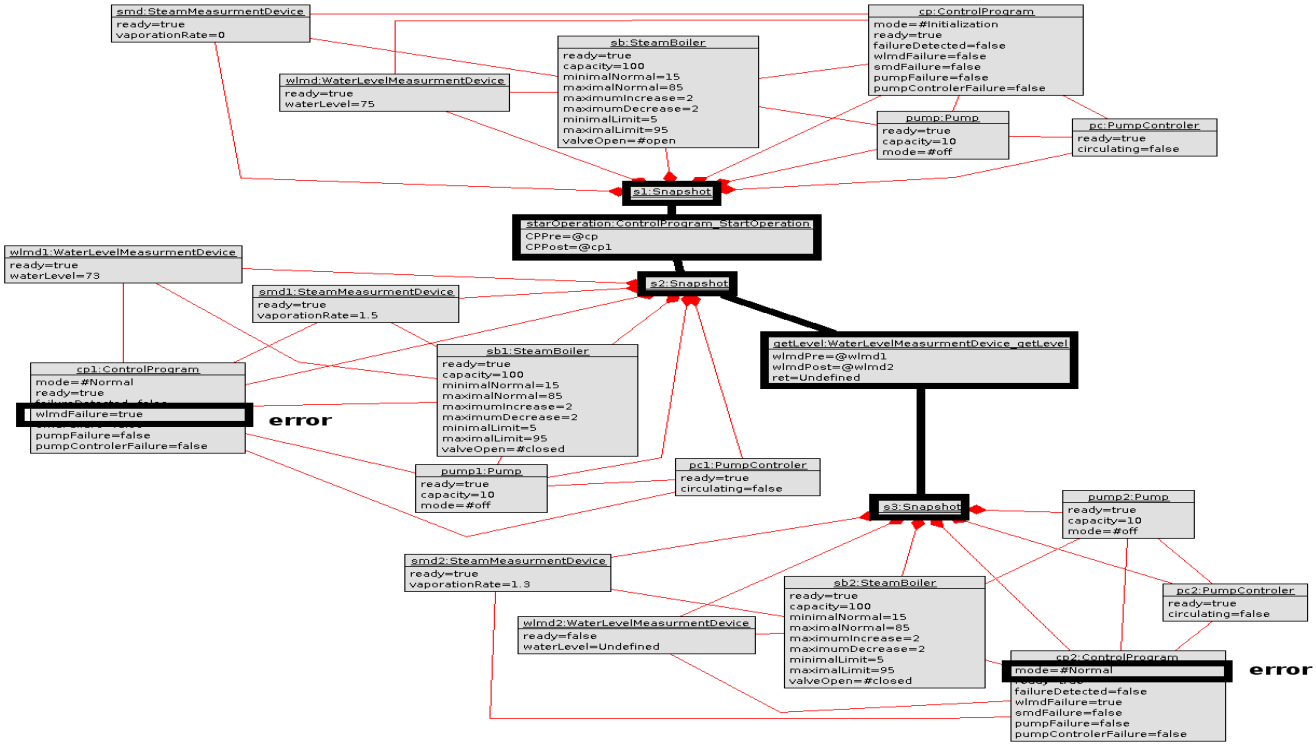

Fig. 4: Counterexample: Scenario violating the temporal property TP1

4.3 Step 3: Analysis

Now, we apply a UML/OCL structural analysis tool to perform the analysis task of the boiler system temporal properties. In this case study, we used the USE Model Validator to check the temporal property expressed in OCL in the steam boiler STM model. For each property, the Validator attempts to produce a scenario (i.e., an instance of the STM) that violates the property. The Validator takes the STM model and an OCL property and provides a relational logic specification. Then the tool employs of-the-shelf SAT solvers to check if there exist an instance of the STM model that violates the OCL expression.

The approach checks a property based on the small-scope hypothesis [10]. That is, when a property does not hold in a model, it is more likely that there is a small scenario that violates the property. Therefore, the approach does not enumerate all possible scenarios, but a constrained number. A scope restricts the number of instances that each class can have in a snapshot and limits the number of transitions in a scenario. As such, the Model Validator enumerates all possible scenarios within the defined scopes and check the given property.

We analyzed the temporal properties in Table 1 on scopes that have one object of each class and 10 transitions. The Validator uncovered a scenario that

Algorithm 1 Snapshot Transitions to Sequence Diagram

Input: Sequence of Snapshot Transition
Output: Sequence diagram
Algorithm Steps: For every object of the Class transition do
Step 1. Get the class name and the operation name that associated with transition.
Step 2. Get the object on which the operation is invoked on.
Step 3. Get the operation parameters from the transition object attributes.
Step 4. Get the return value from the ret attribute of the transition object.
Step 5. Draw a timeline for the object that the operation is invoked on from step 2.
Step 6. Draw an operation invocation on the object using the name of the operation and its attributes from steps 1, and 3 above .
Step 7. Draw a return message of the operation with the value from step 4

violates the first temporal property in Table 1, TP3. Figure 4 shows the counterexample that violates property TP1. To uncover the fault, the verifier must examine the counterexample.

4.4 Step 4: Sequence diagram extraction

The results of Step 3 might be complicated and difficult to present and examine. In this step, we provide support for extracting a sequence diagram from a sequence of snapshot transition. Algorithm 1 provides a systematic way to achieve this objective. We do not show the generated sequence diagram for the lack of space.

5 Related Work

A number of model-checking based techniques have been proposed for specifying and analyzing temporal properties in UML behavioral models, such as statemachines and activity diagrams (e.g., see [3, 4]). In order to apply such techniques, the UML models must be transformed to the tool-specific input languages. For example, the vUML [3] tool automatically converts UML statemachines to PROMELA specifications and then invokes SPIN model checker to verify the desired properties. Although the system is modeled as UML statemachines, the temporal properties are specified in LTL, but not in the UML notation. Eshuis [4] applied symbolic model checking to analyze the data integrity constraints of UML activity diagram and class models. The activity models are formalized and transformed to the input language of the NuSMV model checker. Unlike these techniques, the analysis approach described in this paper neither requires transformation nor requires that the verifier be familiar with notations other than UML and TOCL/OCL.

UML/OCL analysis tools, such as OCLE [6] and USE [11] provide support for validating structural properties. However, OCLE and USE are limited in analyzing temporal properties. The approach described in this paper enables a system designer to analyze TOCL temporal properties using OCLE and USE.

The Scenario-based Design Analysis approach [7] checks whether a given scenario is supported by a design class model. The analysis results depend on the quality of the selected scenarios, which is challenging for complex models . While this approach checks one scenario at a time, the approach in this paper builds on the Scenario-based analysis to check a temporal property within a scope of automatically generated scenarios.

6 Conclusions and Future Work

In this paper, we proposed a lightweight and rigorous approach that uses UML notations for specification and analysis of temporal properties without the need for transformation. The use of TOCL object-oriented temporal logic with specification patterns makes the approach accessible to UML modeling community. As a pointer to future work, we plan to provide a system-development process through which a system designer is able to design complex systems in incremental and iterative manner. Our future work also includes deploying the approach for specifying and analyzing a real-world healthcare Dengue Decision Support System (DDSS) requirements.

References

1. Abrial, J.R., Börger, E., Langmaack, H.: The Stream Boiler Case Study: Competition of Formal Program Specification and Development Methods. In: Formal Methods for Industrial Applications. (1995) 1–12
2. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: Ershov Memorial Conference. (2003) 351–357
3. Lilius, J., Porres, I., Paltor, I.P., Centre, T., Science, C.: vUML: a Tool for Verifying UML Models. (1999) 255–258
4. Eshuis, R.: Symbolic model checking of uml activity diagrams. ACM Trans. Softw. Eng. Methodol. **15** (January 2006) 1–38
5. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling **4** (2005) 2005
6. Chiorean, D., Paşca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML Models Consistency Using the OCL Environment. Electron. Notes Theor. Comput. Sci. **102** (November 2004) 99–110
7. Yu, L., France, R.B., Ray, I., Ghosh, S.: A Rigorous Approach to Uncovering Security Policy Violations in UML Designs. In: ICECCS. (2009) 126–135
8. Al-Lail, M., Abdunabi, R., France, R., Ray, I.: Rigorous Analysis of Temporal Access Control Properties in Mobile Systems. In: ICECCS. (July 2013)
9. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying uml/ocl models using boolean satisfiability. In: MBMV. (2010) 57–66
10. Jackson, D.: Alloy: A Lightweight Object Modeling Notation. ACM Transactions on Software Engneering Methodology **11**(2) (2002) 256–290
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based Specification Environment for Validating UML and OCL. Sci. Comput. Program. **69**(1-3) (2007) 27–34