2016 23rd Asia-Pacific Software Engineering Conference

Model-Based Continuous Verification

Lingling Fan*, Sen Chen*, Lihua Xu*, Zongyuan Yang*, Huibiao Zhu*†

*School of Computer Science and Software Engineering,
East China Normal University, Shanghai, China
†Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
Email: {ecnujanefan, ecnuchensen}@gmail.com, {lhxu, zyyang}@cs.ecnu.edu.cn, hbzhu@sei.ecnu.edu.cn

***Abstract*—Model-based engineering has emerged as a key set of technologies to engineer software systems. While system source code is expected to match with the designed model, legacy systems and workarounds during deployment would undoubtedly change the source code, making the actual running implementation mismatch with its model. Such mismatch poses a challenge of maintaining the conformance between the model and the corresponding implementation. Prior techniques, such as model checking and model-based testing, simply assumed the sole correctness of the model or the implementation, which is naive since they both could contain correct information (e.g. representing either the software requirements or the actual running environment).**

In this paper, we aim to address this problem through model-based continuous verification (ConV), an iterative verification process that links the traditional model checking phase with the software testing phase to a feedback loop, ensuring the conformance between the system model and its implementation. It allows to execute the abstract test cases over the implementation through a semi-automatic binding mechanism to guide the update of the code, and augments system properties from the actually running system to guide the update of the model through model checking. Based on these techniques, we implemented *Eunomia*, a conformance verification system, to support the continuous verification process. Experiments show that *Eunomia* can effectively detect and locate inconsistencies both in the model and the source code.

***Keywords*-consistency checking; model-based testing; linear temporal logic; model checking;**

I. Introduction

Model-based engineering [1, 2] has emerged as a key set of technologies to engineer software systems. Designing and verifying a software system early in its lifecycle, even before the implementation starts, helps to identify problems early and thus prevent software faults from propagating to other development phases [3].

From this traditional verification perspective, the designed model is considered to be the "oracle" during the implementation and maintenance phase. However, there always exist situations where the source code gets updated without strictly following the model, when, for instance, parts of the legacy system are reused or quick fixes must be integrated during deployment. These situations reflect the actual system environment or even additional software requirements that may be ignored during the design phase. Instead of assuming the sole correctness of the designed model, software development is really an iterative process, during which both the model and its source code should be evaluated and updated. More often in industry, we see situations where the designed model and its implementation evolve frequently and concurrently, hence maintaining the conformance becomes more and more challenging over time.

In this work, we propose model-based Continuous Verification (*ConV*), an iterative verification process that intertwines traditional model checking with software testing into a feedback loop. The key to successful and automated support for this iterative process is to first execute the abstract model-based test cases over its implementation, and second retrieve appropriate information from the running system for updating the system model when necessary. To address these challenges, ConV first introduces a semi-automatic binding mechanism to capture the relations between the abstracted model elements and its corresponding implementation, so that the abstracted test cases, which are generated from the system model, can be automatically executed over its implementation with some manual preparation. Secondly, ConV provides a property mining mechanism for transferring the inconsistent properties into Linear Temporal Logic (LTL [4, 5]), and augments it with newly generated properties to guide the update of the system model. And LTL is used as the input to identify the part of the model that showcases the inconsistency. To the best of our knowledge, although model checking has been widely adopted to verify the system model, little effort exists to generate the system properties from implementation.

ConV, as a continuous verification method, can be implemented in different modeling and programming languages. In this paper, built upon the previous work [6], we use Event-B [7, 8] as the modeling language and Java as the programming language, implementing *Eunomia* to fully support the ConV process, and evaluate with several open source systems. The experimental results show that our system can effectively detect and locate the inconsistencies without false alarms, achieving over 88% inconsistency coverage.

In summary, this paper presents the following original

1530-1362/16 $31.00 © 2016 IEEE
DOI 10.1109/APSEC.2016.35

contributions:

- An iterative verification process, linking the model checking phase with software testing phase into a feedback loop to check the conformance between the model and its implementation;
- A semi-automatic binding mechanism to automatically execute abstract test cases over implementation;
- A system property augmentation method based on actual running system, contributing to guide and verify model modification;
- An implemented system named *Eunomia* and its evaluation.

The remainder is organized as follows. In Section II, we provide an overview of our approach. Section III describes *Eunomia*, followed by the evaluation in Section IV. Section V discusses the experimental results and limitations. In Section VI, related studies are presented. Finally, Section VII concludes.

II. ConV – the Iterative Process

In this section, we provide a high-level overview of our model-based continuous verification approach, and the key parts of the iterative process.

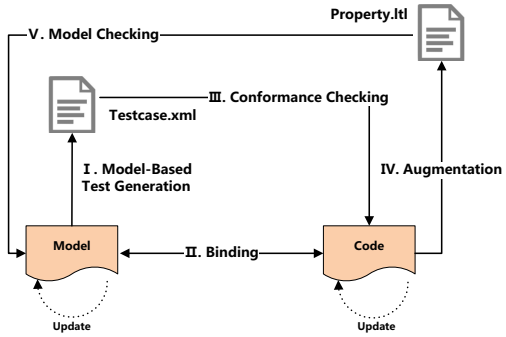


Figure 1. Overview of model-based continuous verification

As shown in Fig. 1, the continuous verification process includes five phases that can be performed iteratively: (i) generating model-based test cases; (ii) binding the abstract model with executable information from the implementation; (iii) testing conformance between the designed model and its implementation; (iv) augmenting the system properties with newly discovered information from the actual running system; (v) model checking with not only the user defined system properties but important information reflecting actual running system.

To help describe ConV, we introduce an illustrative example *mode_system* shown in Fig. 2. It is a module of the Vehicle on Board Control (VOBC) system modeled in Event-B. As shown in Fig. 2, the model of this system is considered as a finite state automaton, where each event has a precondition, and transits to another state when encountering a certain event. The transition model explicitly reveals the possible behaviors of the system. The system events include TRAIN_start (i.e. entering the environment), VOBC_start (i.e. starting the VOBC cycle), VOBC_special (i.e. anticipating inconsistencies between the current mode and change mode), etc. E.g, *q0* is the initial state, when encountering TRAIN_start event, it transits to a new state *q1* where the train is inside the environment of the *mode_system*.

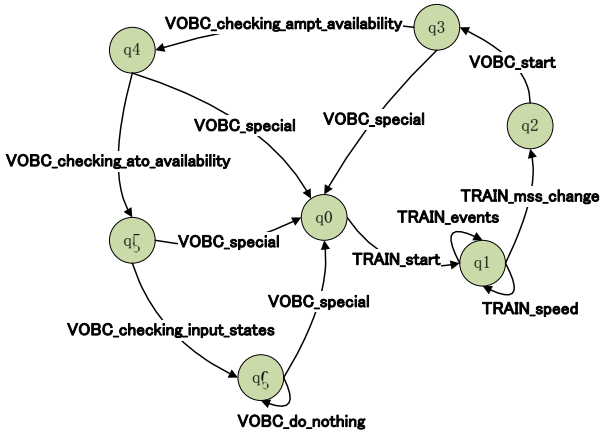


Figure 2. Transition model of *Mode_system*

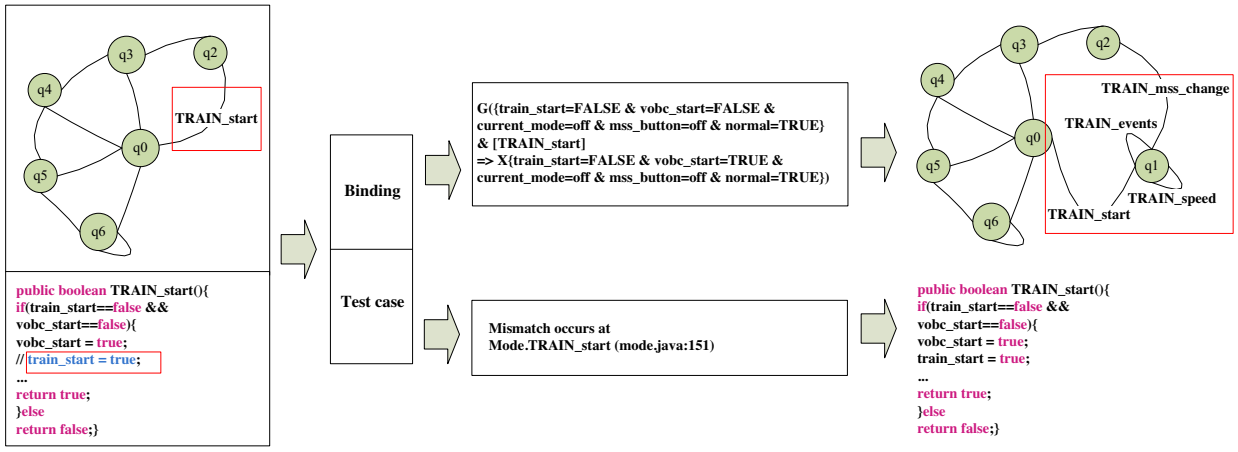
Given an inconsistent model and the source code, ConV is able to provide the bi-directional modification instructions model designers and developers. Fig. 3 illustrates the idea of ConV through the example and the detailed information of test case and binding examples are shown in Fig. 4 and Fig. 5 respectively.

The inconsistent model and code fragment is shown in Fig. 3(a) with a red box. *train_start = TRUE* means the train starts and is in the environment of the mode system, and *vobc_start = TRUE* means the VOBC mode starts, and the train is in the inside of the VOBC mode system[1].

We show how one could utilize ConV to support the bi-directional verification. In traditional model-based testing phase, abstract test cases are generated with respect to the system model. We follow the trend and generate the basic set of test cases to cover the possible behaviors of designed model [9]. Such an abstract test case generated from *mode_system* is shown in Fig. 4. After each event, including initialization, the state is recorded to assist further verification in both directions. For example, the state following INITIALISATION records the initial state of this test case.

Binding. When dealing with the abstract test cases, people

[1]Note that the variables in the original model is *outside* and *inside* respectively, we change them to *train_start* and *vobc_start* only to make it easier to understand for readers.



(a) Inconsistent model & Implementation (b) Binding & Test case (c) Conformance checking & Property augmentation (d) Updated model & Implementation

Figure 3. An illustrative example of ConV through *mode_system*

```
<test_case ID="1">
   <INITIALISATION>
      <value type="variable">FALSE</value>
      <value type="variable">off</value>
      <value type="variable">1</value>
      ...
   </INITIALISATION>
   <state vobc_start="FALSE" train_start="FALSE".../>
   <event name="TRAIN_start"/>
   <state vobc_start="FALSE" train_start="TRUE".../>
   <event name="VOBC_start"/>
   <state vobc_start="TRUE" train_start="TRUE".../>
   ...
</test_case>
```

Figure 4. An abstract test case example

```
1  public class ModeMapping implements ModelMapper {
2    private Mode m = Mode.getInstance();
3    // Bind Event (model) to Function (code)
4    public boolean executeEvent(Event event){
5       if(event.getName().equals("TRAIN_start")){
6       return m.TRAIN_start();
7     }else if(event.getName().equals("TRAIN_mss_change")){
8       return TRAIN_mss_change();
9     }else if
10        ...
11    else{
12      System.out.println("Unknown event:"+event.getName());
13      return false;
14    }
15   };
16   // Bind State (model) to Variable (code)
17   public State getCurrentState(){
18     State state = new State();
19     if(m.getoutside()==machine.Mode.BooleanType.TRUE){
20       state.getVariableValues().put("vobc_start", "TRUE");
21     }else{
22       state.getVariableValues().put("vobc_start", "FALSE");
23     }
24        ...
25     return state;
26   };
27 }
```

Figure 5. A binding example

can hardly find a universal solution for all targeted real systems. To alleviate this problem, we propose a semi-automatic binding mechanism to bridge the gap between the behaviors of the model and the functions of the implementation through templates, as shown in Fig. 5. It provides a template implementing the `ModelMapper` interface to map Events and States in terms of the specific source code. The `ModelMapper` consists of two method declarations: `eventMapper` and `stateMapper`.

- **eventMapper.** As for the `eventMapper`, users are required to change the Event names (i.e. arguments in the `eventMapper` class) of the model and the corresponding function names of the source code in the template. When executing, the template returns the corresponding functions for the source code execution. If they are not correctly mapped, it throws a exception: *Unknown event detected*, so as to guide the users to modify the binding process.
- **stateMapper.** The `stateMapper` requires users to change the variables and the corresponding values of the model to the variables declared in the source code. When executing, the template not only returns the variable values that benefits the execution over abstract test cases, but it also outputs the execution results of the system states used to augment properties later. The template is able to deal with different data types, such as Boolean, Integer and String. In terms of different data types, it provides different forms of templates.

The template defines the relations between the two artifacts. When executing abstract test case, the state and event information will be automatically extracted and executed over the corresponding source code, with respect to user-

defined relations. As in the example *mode_system*, the code fragment shown in Fig. 3(b) is part of the core of binding. It maps event *TRAIN_start* and variable *vobc_start* of the model to function *TRAIN_start()* and variable *vobc_start* of the source code, respectively.

Conformance Testing. As one of the key parts of *ConV* process, it is important to analyze and maintain the conformance between the system model and its implementation. The conformance testing phase aims to analyze the system behaviors under the premise of the given input (i.e. test case), checking the executing results against the test cases generated from the model to detect the mismatches between both artifacts.

Specifically, after binding the transitions and states, the abstract test cases can be automatically executed over the implementation. For each test case, if all the transitions are run successfully and the states are shown as expected, the test case is then passed; otherwise, there are two possible reasons the test case failed: the state change is different from the state of original test case or one transition cannot be executed at all as its precondition cannot be satisfied. No matter whether the test case is passed, execution results are recorded, as well as the mismatch location where implementation violates the designed model shown in Fig. 3(c).

System Property Augmentation. In ConV, inconsistent system information is extracted from the actual running implementation, and transformed into LTL, improving the completeness of system properties in model checking. The LTL generation process is as follows.

Algorithm 1 LTL Generation Algorithm

Input: Abstract test cases, execution result
// denoted as absTc and exeRe respectively
Output: LTL

1: $LTL \leftarrow \{\}$; // initialize LTL with empty
2: **for** each $tc1 \in absTc$ and each $tc2 \in exeRe$ **do**
3: **if** $tc1.id$ == $tc2.id$ **then**
4: compare the values of each variable;
5: **if** ! $compare$(tc1, $tc2$) **then**
6: $LTL \leftarrow LTL \cup tc2.get(propeties)$;
7: **end if**
8: **end if**
9: **end for**
10: **return** LTL

As shown in Algorithm 1, the input consists of the abstract test cases and its execution results, and the output is the inconsistencies represented in LTL. Firstly, LTL is initialized with empty (line 1). For each test case, we compare the states of each test case with execution results. If mismatch occurs, the inconsistent property will be transformed to LTL (line 6). The inconsistent semantics in the example can be extracted and represented in LTL shown in Fig. 3(c). It means that the model should be in the state where *train_start* = *false* after the event TRAIN_start.

The resulting properties are often redundant, since each *State* consists of several variables, and each *Event* might also consist of several arguments. To alleviate this problem, we propose an LTL optimization algorithm as depicted in Algorithm 2.

Algorithm 2 LTL Optimization Algorithm

Input: Original LTL, IS // IS:InconsistencySet
Output: Optimized LTL

1: $VR \leftarrow \{\}$; // initialize VR with empty
// VR:VariableRange containing all variables and corresponding value spaces.
2: $currentRange \leftarrow \{\}$;
3: **for** each $variable \in TestCases$ **do**
4: $VR \leftarrow VR \cup (variable, variable.values)$;
5: **end for**
6: **for** each $var \in VR$ **do**
7: $originalRange \leftarrow VR.get(var)$;
8: **for** each $incon \in IS$ **do**
9: $currentRange \leftarrow currentRange \cup incon.get(var)$;
10: **end for**
11: **if** $originalRange$ equals $currentRange$ **then**
12: $InconLeft \leftarrow IS \setminus var$;
13: Classify $InconLeft$ by each value in $originalRange$
14: **if** classified $InconLeft$ equals each other **then**
15: $IS \leftarrow IS \setminus var$;
16: **end if**
17: **end if**
18: **end for**
19: **return** IS

Algorithm 2 takes the original LTL as input, and outputs the optimized LTL. Let *IS* be the Inconsistency Set represented in LTL, and *VR* be the Variable Ranges to record all variables and corresponding value spaces. *VR* is initialized empty (line 1) and filled after traversing all the test cases to obtain the value space of each variable (line 4). *currentRange* refers to the value space obtained from *IS*.

For each variable in *VR*, we first extract the value space from the *VR* (line 7) denoted *originalRange*, then extract the value space of the each variable from LTL denoted as *currentRange* (line 9). The two sets are then compared (line 11). If *currentRange* does not equal *originalRange*, which means not all the values of the variable are taken into consideration in the analysis, the uncovered value may lead to other effects. Thus this variable under discussion should not be deleted from the LTL. Otherwise, we define the rest of inconsistency set that excludes *var* as *InconLeft* (line 12), and group it in terms of *var* (line 13). If the classified *InconLeft* equals to each other (line 14), which

means the excluded variable (i.e. *var*) has no effect on the LTL. Thus *var* can be deleted from *IS* (line 15). Finally *IS* is reduced after combining the same inconsistencies.

We illustrate it with a simple example shown in Fig. 6. The final LTL sequence contains two LTLs. We first extract the value space of each variable, for example, the value space of *normal* is {TRUE, FALSE}, which equals to the original value space of *normal* (i.e., *normal* has no other values in the whole program). The rest of the LTL that excludes *normal* is exactly the same, indicating that *normal* has no effect on this LTL and can be deleted.

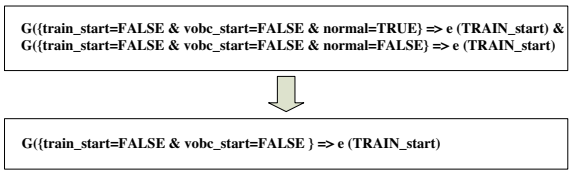


Figure 6. A LTL optimization example

Furthermore, these newly discovered properties are augmented iteratively. Throughout the continuous verification process,the model consistent with the source code is shown in Fig. 3(d).

III. Implementation

The idea of *ConV* can be applied to most formal languages, we use Event-B and Java, implementing *Eunomia* to fully support ConV, since Event-B can ensure the correctness of design without ambiguity.

As shown in Fig. 7, *Eunomia* is built upon extended Event-B MBT [8] and ProB [10]. Event-B MBT is a model-based test case generator, we extend it with execution to not only record the previous states of the events, but also the post states. ProB is a model checker for B method. *Eunomia* consists of three other main components: *TCExecutor*, *LTLGenerator* and *LTLOptimizer*. Components communicate through XML files or LTL files, reducing the coupling effect and increasing the flexibility of the system.

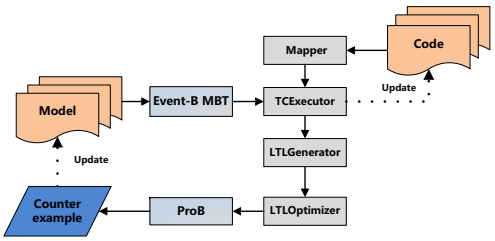


Figure 7. System architecture of *Eunomia*

TCExecutor takes as input the abstract test cases, outputs the passed test cases in the same xml format, as well as the mismatch location in the source code. And *LTLGenerator* takes as input from *TCExecutor*, extracts system properties and outputs the inconsistent behaviors represented in LTL. It represents each inconsistency as a dedicated LTL, using "&" to join all the LTLs, which can be directly fed to the model-checker. *LTLOptimizer* takes as input the original LTLs, and automatically combines the LTLs and deletes invalid variables, outputs the optimized LTLs. The LTLs (the original or the optimized, depending on user's choice) are fed to ProB as input to check the model, and output counterexamples to show the location of the mismatch. Thus designers can update the model to comply with the code.

IV. Empirical Evaluation

Our experimental study is designed to answer the following research questions:

- **RQ1:** *How does ConV perform in identifying inconsistencies?* This research question is to investigate whether ConV is able to identify the inconsistencies existing in the system.
- **RQ2:** *How do different levels of abstraction of the system model influence the effectiveness of ConV?* This research question is to investigate the abstraction level that ConV fits for.
- **RQ3:** *How does ConV prove that the identified inconsistencies are accurate and valid?* This research question is to investigate whether ConV identifies inconsistencies accurately.
- **RQ4:** *How does ConV perform in locating the identified inconsistencies?* This research question is to investigate whether ConV is able to locate the identified inconsistencies, and provide modification instructions.

A. Experimental Setup

Although the proposed approach can apply to different formal languages, *Eunomia* is language-dependent. Thus we conduct experiments on three Event-B models, available publicly at Lab301[2]. We choose a water_tank model, a water_boiler model and a mode_system model. These models are well validated in academia [11].

To evaluate the inconsistency-checking ability of ConV, we conduct experiments on different models of different scales. For setting up the inconsistency situation, we first insert some inconsistencies into the source code, then check the conformance with *Eunomia*. To avoid bias, we enumerate through all the methods of the source code, and randomly select the methods that should be inserted inconsistencies.

Additionally, We believe that ConV should work with system models in different levels of abstraction. To evaluate this, we further carry out experiments on the 9 refined models of the mode_system, which are treated as 9 separated systems, named *Mode0* to *Mode8*. The inconsistencies are inserted randomly again into their respective versions of

[2]http://www.lab205.org/home/#!/hybrid-eventb

implementation. Furthermore, we also evaluate *Eunomia's* inconsistency locating capability. The interesting results from this experiment setup cross-confirm the validity of the inconsistency-checking ability from another aspect since the inconsistencies are accurately revised according to *Eunomia*.

B. RQ1: Inconsistency-checking ability

The goal of this study is to evaluate the inconsistency-checking ability of *Eunomia*. Table I shows the results of extracting the inconsistencies of each system. The *TCs* column lists the number of test cases generated from each model using Event-B MBT. The *Incons S* column describes the inconsistencies inserted into the source code and the *Incons I* column gives the number of identified inconsistencies, and the last column shows the ratio of the identified inconsistencies. For example, with regard to mode_system in Table I, 192 test cases are generated from the model, and 60 inconsistencies are inserted into the source code, 55 of which have been identified, achieving 91.67% inconsistency coverage.

Table I
INCONSISTENCY-CHECKING ABILITY ON DIFFERENT SYSTEMS

Models	TCs	Incons S	Incons I	% of Incons identified
mode_system	192	60	55	91.67
water_boiler	18	23	21	91.30
water_tank	17	20	18	90.00

C. RQ2: Cross-abstraction inconsistency-checking ability

Table II shows the results of evaluation on models in different abstract levels. The notations share the same meaning as those in Table I. The data indicates that the percentage of identified inconsistencies for all the models ranges from 88.89% to 96.67%. Since design models do not include all necessary information to generate fully functional implementations, code can be defined beyond model definition where *Eunomia* may miss some inconsistencies. Overall, the data in Table I and Table II show that *Eunomia* can efficiently detect the inconsistencies between model and code. Even in the worst case, *Eunomia* can achieve over 88% of the inconsistency coverage.

Table II
CROSS-ABSTRACTION INCONSISTENCY-CHECKING ABILITY

Abstract levels	TCs	Incons S	Incons I	% of Incons identified
Mode0	8	9	8	88.89
Mode1	9	10	9	90.00
Mode2	18	20	18	90.00
Mode3	37	34	31	91.18
Mode4	31	30	29	96.67
Mode5	47	40	36	90.00
Mode6	97	46	43	93.48
Mode7	302	55	51	92.73
Mode8	192	60	55	91.67

During the experiment, we analyzed and compared the results of Mode0 to Mode8 and found an interesting phenomenon that two of the inconsistencies led to the same location of the original model. Through further investigation, we confirmed that the two inconsistencies were introduced in the first layer of the model and were propagated to its refinements. This phenomena further confirms Eunomia's effectiveness of identifying inconsistencies at different levels of abstraction.

D. RQ3: Validity of the identified inconsistencies

The goal of this study is to evaluate the validity of the inconsistencies identified by *Eunomia*. We carefully designed two inconsistencies that we introduce to the system. These two inconsistencies meet the following two requirements: (i) They can be introduced in the most abstract level; and (ii) they are able to propagate to the following levels. Our hypothesis is that an inconsistency identification mechanism should be able to catch these two consistencies at every level of abstraction, and the two inconsistencies should point to the same location. Therefore, we inserted the two special inconsistencies and six random inconsistencies into the first layer of the model, focusing on the number of inconsistencies that continues existing from the original model to the final one. We treat them as the same ones since they lead to the same modification locations in the model and the code.

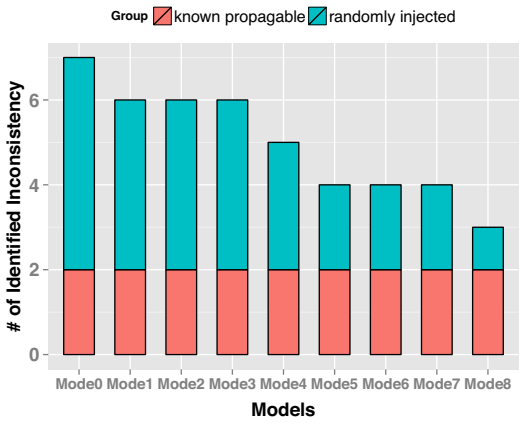


Figure 8. Validity of the identified inconsistencies

As we can see in Fig. 8, the *Models* lists each refined model of Mode_system named *Mode0* to *Mode8*. Note that the inserted inconsistencies consist of the two special ones and six randomly inserted ones, and we only insert them to the most abstract model. The data in Fig. 8 indicate that not all the inserted inconsistencies have been identified, and the number of identified inconsistencies in each system is not equal. Seven of them are identified by *Mode0*, while only three of them are propagated and identified by *Mode8*. Moreover, it decreases with the refining of the model.

Note that each system has identified the two special inconsistencies and a newly discovered one. And we have manually verified that the inconsistencies are represented in

LTL of the same semantics, indicating the three (2+1) identified inconsistencies are indeed the same ones, which proves the validity of identified inconsistencies. We further analyze other inconsistencies that are missing, and the reasons will be discussed in Section V.

E. RQ4: Inconsistency-locating capability

The goal of this study is to evaluate the inconsistency locating capability of *Eunomia*. To carry out the experiment, We focus on the three identified inconsistencies of Mode_System in *RQ3*, which are represented as LTLs of the same semantics. We ran the model checker to locate the inconsistencies and reran *Eunomia* to verify the modification.

Fig. 8 has shown that it is able to detect the inconsistencies, thus, we either modify the models according to the counterexamples displayed in model checker, or modify the source code according to the location *Eunomia* provides. To verify whether our modification correctly keeps the conformance between the model and source code, either the model checker or *Eunomia* re-check the updated ones. As we expected, the inconsistencies detected by *Eunomia* were eliminated.

In short, our continuous verification process supported by *Eunomia* can efficiently locate the inconsistencies both in the model and its implementation.

V. Discussion

Reasons for missing behaviors. As shown in Table I and Fig. 8, *Eunomia* missed some inconsistencies. *Eunomia* partly depends on the efficiency of Event-B MBT since the test cases are generated from Event-B MBT. Unfortunately, as many other testing tools, test generation suffers from infeasible paths and state space explosion. There are possibilities that the inserted inconsistencies exist in the infeasible paths of implementation that are not covered by the generated test cases. As a result, test cases are passed over the implementation without covering such inconsistencies, *Eunomia* then reports that no inconsistencies are found. For example, in the mode_system, if there is an inconsistency in the event *VOBC_do_nothing* that sets *train_start = true*, and we use Event-B MBT to generate test cases from model as usual. However, the generated test cases cannot cover the event *VOBC_do_nothing* due to the limitation of Event-B MBT, and the test case are all passed without executing the function of *VOBC_do_nothing()* in the code. As a result, *Eunomia* misses such inconsistencies.

Another possible reason for missing behaviors is due to the different abstraction of system model vs. its implementation. Specifically, test cases are generated from the abstract model, hence inconsistencies maybe beyond pre-description in the model. Under such circumstance, the generated test cases will never violate the source code. As a result, the generated test cases will all pass when executed over the code, leading to the fact that there is no difference between the test cases and the execution results.

Interesting phenomenon. As shown in Fig. 8, the number of identified inconsistencies decreases. It is because the inconsistencies that are inserted into the previous versions may be restored during the refinement process. For example, we set the variable *train_start = false*, which was *true* originally in the event *VOBC_Start* in Mode0. While Mode 4 redefined that *train_start = true*, which is conformable with the source code. As a result, *Eunomia* reports that there is no inconsistency between Mode4 and its implementation.

The limitations. The system that *Eunomia* can handle depends on the number of events and variables in the model, rather than the lines of source code. Because the number of execution paths is determined by events and variables, which directly affects the time cost of test case generation.

There are 47 events and 36 variables in the Mode_system, and *Eunomia* successfully handled it. Thus, although more experimentations will help further find out the capability of *Eunomia*, we believe that *Eunomia* is likely to handle more complex systems.

VI. Related Work

Conformance Checking. Heidenreich et al. [12] proposed the *JaMoPP* approach to bridge the gap between modeling languages and programming language Java. It treats source code as models, and can transfer it to other programming languages. However, this approach does not take the construction or the transfer process into consideration.

DiaSpec [13] introduced *interaction contrasts* to guarantee conformance between the architecture and its implementation by generating dedicated programming framework. It can re-generate framework without overwriting the code, but only support one-way changes of the architecture or the code.

Czepa et al. [14] proposed plausibility checking of LTL-based specifications. It can help users create the new constraint patterns and provide confidence through the LTL representation. However, it cannot check the conformance between the model and the source code.

Property Mining. Riedl-Ehrenleitner et al. [15] proposed an approach that can mine the invariants from the code and check them on the model to validate the efficiency. This approach may cause difficulty in some complex code analysis, requiring the high-conformance of Objects and Functions in both model and code.

Su and Gabel et al. proposed *DejaVu* [16], a highly scalable system for detecting these general syntactic inconsistency bugs between different source code versions. They can automatically check whether there exist the defined inconsistencies. However, they focus on mining the properties between different code versions.

Mapping. Zheng et al. [17] proposed the *1.x-way architecture-implementation mapping* approach, which help-

s to manage changes and map the behavior architecture specification to the implementation. However, this approach maintains the conformance of the model and the code by generating code.

Ubayashi et al. proposed *Archface* [18], based on *component and connector* architecture [19], performs a programming-level interface mechanism to bridge the gap between the architecture and its implementation. However, *Archface* is only designed for AOP [20] language and relies on a dedicated compiler to detect mismatches.

VII. Conclusions

In this paper, we have proposed model-based continuous verification, linking the traditional model checking and software testing into a conformance checking feedback loop. Our approach generates and executes test cases from the system model when it is considered to be appropriate; and augments system properties, when updating system model is required, to represent the actual running implementation. Furthermore, we implemented the approach in *Eunomia*, and verified it on several open source systems. Experiment results show that *Eunomia* can efficiently detect and locate the inconsistencies both in the model and the source code.

Acknowledgment

This work was supported by the National Natural Science Foundation of China, under Grant 61502170 and National Natural Science Foundation of China (Zhong Dan) International Cooperation Project, under Grant 61361136002.

References

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 0025–31, 2006.

[2] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Software Engineering*, vol. 1, no. 1, pp. 1–182, 2012.

[3] B. Nuseibeh, S. Easterbrook, and A. Russo, "Leveraging inconsistency in software development," *Computer*, vol. 33, no. 4, pp. 24–29, 2000.

[4] K. Y. Rozier, "Linear temporal logic symbolic model checking," *Computer Science Review*, vol. 5, no. 2, pp. 163–203, 2011.

[5] S. Babenyshev and V. Rybakov, "Linear temporal logic ltl: basis for admissible rules," *Journal of Logic and Computation*, p. exq020, 2010.

[6] X. Ge and L. Xu, "Prominer: Bi-directional consistency checking framework based on system properties," 2016.

[7] J.-R. Abrial, *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[8] I. Dinca, F. Ipate, L. Mierla, and A. Stefanescu, "Learn and test for event-b–a rodin plugin," in *Abstract State Machines, Alloy, B, VDM, and Z*. Springer, 2012, pp. 361–364.

[9] I. Schieferdecker, "Model-based testing," *IEEE software*, no. 1, pp. 14–18, 2012.

[10] M. Leuschel and M. Butler, "Prob: A model checker for b," in *FME 2003: Formal Methods*. Springer, 2003, pp. 855–874.

[11] W. Su, J.-R. Abrial, and H. Zhu, "Complementary methodologies for developing hybrid systems with event-b," in *International Conference on Formal Engineering Methods*. Springer, 2012, pp. 230–248.

[12] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the gap between modelling and java," in *Software Language Engineering*. Springer, 2010, pp. 374–383.

[13] D. Cassou, E. Balland, C. Consel, and J. Lawall, "Leveraging software architectures to guide and verify the development of sense/compute/control applications," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 431–440.

[14] C. Czepa, H. Tran, U. Zdun, T. T. K. Tran, E. Weiss, and C. Ruhsam, "Plausibility checking of formal business process specifications in linear temporal logic," in *CAISE*, 2016.

[15] M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed, "Towards model-and-code consistency checking," in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*. IEEE, 2014, pp. 85–90.

[16] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, "Scalable and systematic detection of buggy inconsistencies in source code," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 175–190.

[17] Y. Zheng and R. N. Taylor, "Enhancing architecture-implementation conformance with change management and support for behavioral mapping," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 628–638.

[18] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: A contract place where architectural design and code meet together," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 75–84.

[19] R. Allen and D. Garlan, "Formalizing architectural connection," in *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 71–80.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European conference on object-oriented programming*. Springer, 1997, pp. 220–242.