Counterexample Analysis for Supporting Containment Checking of Business Process Models

Faiz UL Muram, Huy Tran, Uwe Zdun

Software Architecture Group
University of Vienna, Austria.
faiz.ulmuram|huy.tran|uwe.zdun@univie.ac.at

Abstract. During the development of a process-aware information system, there might exist multiple process models that describe the system's behavior at different levels of abstraction. Thus, containment checking is important for detecting unwanted deviations of process models to ensure a refined low-level model still conforms to its high-level counterpart. In our earlier work, we have interpreted the containment checking problem as a model checking problem and leveraged existing powerful model checkers for this purpose. The model checker will detect any discordance of the input models and yield corresponding counterexamples. The counterexamples, however, are often difficult for developers with limited knowledge of the underlying formal methods to understand. In this paper, we present an approach for interpreting the outcomes of containment checking of process models. Our approach aims to analyze the input models and counterexamples to identify the actual causes of containment inconsistencies. Based on the analysis, we can suggest a set of countermeasures to resolve the inconsistencies. The analysis results and countermeasures are visually presented along with the involved model elements such that the developers can easily understand and fix the problems.

Keywords: Counterexample analysis, containment checking, consistency checking, BPMN, process model, behavior model, model checking, countermeasure.

1 Introduction

Model checking is a powerful verification technique for detecting inconsistencies of software systems [8]. In general, especially in the context of business process management, behavior models are transformed into formal specifications and verified against predefined properties. The model checker then exhaustively searches for property violations in formal specifications of a model and produces counterexample(s) when these properties do not satisfy the formal specifications. The ability to generate counterexamples in case of consistency violations are considered as one of the strengths of the model checking approach. Unfortunately, counterexamples produced by existing model checkers are rather cryptic and verbose. In particular, there are two major problems in analyzing counterexamples. First, the developers and non-technical stakeholders who often have limited knowledge of the underlying formal techniques are confronted with cryptic and lengthy information (e.g., states numbers, input variables over tens of cycles and internal transitions, and so on) in the counterexample [11]. Second, because a counterexample is produced as a trace of states, it is challenging to trace back the causes of inconsistencies to the level of the original model in order to correct the flawed elements [9]. As a result, the developers have to devote significant time and effort in order to identify the cause of the violation, or they get confused about the relevance of a given trace in the overall explanation of the violation. Besides that, in order to raise the practical applicability of model checking, there is a need for an automated approach to interpret the counterexamples with respect to containment checking and finding the causes of a violation.

There is a certain amount of approaches that target counterexample analysis for model checking [4, 9, 11]. Out of these existing approaches, only a few are aiming at supporting counterexample analysis of behavioral models [11]. Most of these approaches focus on fault localization in source programs for safety and liveness properties or generation of proofs to aid the understanding of witnesses. As these approaches focus on model checking in a generic context, their analysis techniques can be applied in a wide range of application domains. However, this comes with a price: the analysis outcomes are rather abstract and far from helping in understanding the specific causes of a violation in a particular domain. Furthermore, these techniques have not considered to provide any annotations or visual supports for understanding the actual causes nor suggest any potential countermeasures

In this paper, we propose to focus on a specific context, which is the *containment checking problem*, in order to achieve better support for understanding and resolving the inconsistencies. The goal of containment checking is to ensure that the specifications in high-level models created, for instance, by business analysts early in the development lifecycle, are not unwittingly violated by refined (one or more low-level) models of the high-level model created, for instance, by developers during the detailed design phase [13]. The containment checking problem can be interpreted as a model checking problem, in which the behavior described in the high-level model are used as the constraints that the low-level counterpart must satisfy [13]. In case of a containment violation, the model checkers will generate corresponding counterexamples.

We have developed an approach that supports the interpretation of the generated counterexamples and reports typical possible causes of a containment inconsistency. In particular, we have constructed a counterexample analyzer that automatically extracts the information from the counterexample trace file generated by containment checking using the NuSMV model checker [5]. Based on the extracted information along with formalization rules for the containment relationship, our counterexample analyzer identifies the cause(s) of the violation(s) and produces an appropriate set of guidelines to countermeasure the containment violations. Our approach allows the developers to focus on the immediate cause of an inconsistency without having to sort through irrelevant information. In order to make our approach more usable in practice, we devise visual supports that can highlight the involved elements in the process models. Furthermore, it provides annotations containing causes of inconsistencies and potential countermeasures shown in the input process models. In the scope of this study, we consider BPMN [1] process models because they are widely used for the description of business processes and the majority of process developers are familiar with their syntax and semantics.

The paper is structured as follows. In Section 2, we provide background information on our model-based containment checking approach. Section 3 describes the counterexample interpretation approach in detail. Section 4 presents a use case extracted from an industrial case study to illustrate our approach. In Section 5, we review the related approaches regarding behavioral consistency checking and counterexamples interpretation. Finally, we conclude on our main contributions and discuss future work in Section 6.

[1] `http://www.omg.org/spec/BPMN/2.0`

2 Model Checking Based Containment Checking Approach

This section briefly introduces important aspects of our model checking based approach for containment checking [13]. Containment checking aims to verify whether the elements and structures (e.g., activities, events and/or gateways) of a high-level BPMN model correspond to those of a refined low-level model. An example of a high-level model is a specification produced by a business analyst together with a customer early in the software development lifecycle. Later during architecting and detailed design, this model is usually gradually refined for subsequent implementation. Containment checking can ensure that the high-level behavior model is completely contained in the refined models and that developers did not (unwittingly) introduce deviations (aka consistency violations). According to the definition of containment relationship, the opposite direction is not essential because the low-level behavior models are often constructed by refining and enriching the high-level model. An overview of the approach is shown in Figure 1.

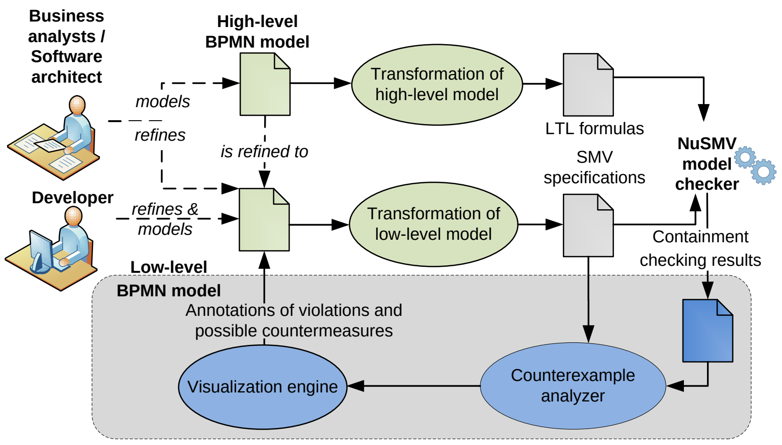
Business analysts / Software architect
High-level BPMN model
Transformation of high-level model
models
refines
is refined to
LTL formulas
NuSMV model checker
SMV specifications
Developer
refines & models
Transformation of low-level model
Containment checking results
Low-level BPMN model
Annotations of violations and possible countermeasures
Visualization engine
Counterexample analyzer

Fig. 1: Overview of the containment checking approach

As shown in Figure 1, the high-level process model under study will be automatically transformed into Linear Temporal Logic (LTL) rules [14] whilst the low-level counterpart will be automatically transformed into a SMV specification (input language of NuSMV model checker) [6]. Then, containment checking is performed by leveraging the NuSMV model checker [5]. In particular, the model checker takes the generated LTL rules and the SMV specification as inputs to verify whether they are satisfied. In case the SMV specification satisfies the LTL rules, it implies that the low-level process model conforms to the corresponding high-level model. If, however, the low-level process model deviates improperly from the high-level counterpart, NuSMV will generate a counterexample that consists of the linear (looping) paths of the SMV specification leading to the violation. The main focus of this paper is shown in the big grey box.

The counterexample essentially shows the progress of the states from the beginning (i.e., all variables are initialized) until the point of violation along with the corresponding variables' values. Hence, it is time consuming and error-prone to locate relevant states because the developers may have to exhaustively walk through all of these execution traces. We note that counterexamples generated by the NuSMV model checker may contain different information depending on the selected model checking options, model encoding techniques, and the input LTL rules. Thus, it is crucial to provide useful feedbacks to the developers and non-technical stakeholders that can reveal the causes of containment inconsistencies and suggest potential resolutions.

In the subsequent sections, we present in detail our approach for counterexample interpretation that is able to help identifying the causes of containment inconsistencies. It also provides an appropriate set of guidelines to developers containing countermeasures to address the deviations from the containment relationship. The low-level model is updated based on the guidelines and re-mapped to its formal specification, and then will be re-verified. This process iterates until no containment violations are detected.

3 Interpretation of Containment Inconsistencies

Containment inconsistencies may occur due to a variety of reasons, such as missing or misplaced elements in the low-level model, and so on. We propose a two-step approach for locating the causes of containment inconsistencies. In the first step, the counterexample analyzer extracts the information from the output trace file generated by the NuSMV model checker and identifies the actual causes of the unsatisfied containment relationship and produces appropriate suggestions. In the second step, the information provided by the counterexample analyzer will be annotated in the low-level process model along with concise descriptions of the violation's causes and potential countermeasures.

3.1 Counterexample Analyzer for Locating Causes of Containment Inconsistencies

The counterexample analyzer investigates the causes of an unsatisfied containment relationship with respect to the LTL-based primitives. Initially, the counterexample analyzer reads the output trace file and parses the counterexamples that represent the unsatisfied LTL rules. Afterwards, the counterexample analyzer traverses the extracted information, LTL-based primitives and SMV specification to find out why the elements and control flow structures of the high-level model are not matched by their corresponding low-level counterparts. Note that the elements that are described in the high-level model but missing or misplaced in the low-level model can be the causes of the containment inconsistencies. The counterexample analyzer inspects and addresses all possible causes of an unsatisfied containment relationship defined by a specific LTL-based primitive and possible countermeasures.

In order to locate the causes of the inconsistency, the counterexample analyzer first verifies whether all the elements (e.g., activities, events and/or gateways) that exist in the high-level model are also present in the low-level model. For this, the counterexample analyzer locates the missing element cause (either one, multiple, or all elements could be missing) and suggests the countermeasure (i.e., insert missing element at a specific position in the model).

After that a number of rules related to unsatisfied LTL rules for different possible kinds of elements in the BPMN model are checked. For this, the counterexample analyzer matches the exact position of the corresponding elements in the high-level model related to unsatisfied LTL rules with elements present in the low-level model. Specifically, the counterexample analyzer reads the sequence (of elements of the low-level model) from the SMV specification

Table 1: Tracking back the causes of containment violations and relevant countermeasures

Elements	**LTL-Based Primitives**	**Causes of Unsatisfied Rule**	**Possible Countermeasures**
Sequence: A set of elements (transitively) executed in sequential order.	`(G (A1 -> F A2))`	The sequential rule is violated, if element A2 does not eventually follow element A1 in the low-level model, but A2 exists as a preceding element of A1.	• Swap the occurrence of A2 and A1. • Add A2 after A1 in the low-level BPMN model.
Parallel Fork (AND-Split): The execution of a Fork leads to the parallel execution of subsequent activities or events (A1, A2...An). Please note that the activities or events may be executed one after the other or possibly may be executed in a real parallel enactment.	`G (ParallelFork -> F (A1 & A2 &...& An))`	The Parallel Fork rule is unsatisfied, if a Fork gateway is not eventually followed by either one or all the activities/events (A1, A2,...An), or either one or all the activities/events exist as a preceding element of a Fork gateway.	• Put elements (A1 and/or A2 ...and/or An) after the Parallel Fork in the low-level model. • Elements (A1, A2...An) shall be triggered from the Parallel Fork.
Parallel Join (AND-Join): The execution of two or more parallel elements (A1, A2...An) leads to the execution of a Join gateway. The semantics is represented that all elements must complete before the execution of a Join gateway.	`(G (A1 & A2 & ...& An) -> F ParallelJoin)`	The Parallel Join rule is violated, because either one or all the elements (A1, A2 ... An) exist as succeeding elements of a Join gateway, but are not followed by a Join gateway.	• Replace flawed elements(s) (“element’s name”) with the correct elements (“element’s name”), respectively. • Remove flawed element(s) (“element’s name”) from the low-level BPMN model. • Elements (A1, A2...An) shall be followed by a Parallel Join.
Exclusive Decision (XOR-Split): The execution of an Exclusive Decision eventually followed by the execution of at least one of the elements among the available set of elements based on condition expressions for each gate of the gateway.	`(G (ExclusiveDecision -> F (A1 xor A2)))`	The Exclusive Decision rule is violated, if both of the branches return either FALSE or TRUE exclusively. It means that the Exclusive Decision gateway is not followed by elements (i.e., A1 and A2).	• Replace flawed elements (“element’s name”) with correct elements (“element’s name”) after the Exclusive Decision, respectively. • Remove flawed elements (“element’s name”) from the low-level BPMN model.
Exclusive Merge (XOR-Join): The execution of at least one element among a set of alternative elements will lead to the execution of an Exclusive Merge gateway.	`( G (A1 xor A2) -> F Exclusive-Merge)`	The Exclusive Merge rule is unsatisfied, because activity A1 and activity A2 are not followed by an Exclusive Merge, but one or both elements exist as the succeeding elements of an Exclusive Merge in the low-level model.	• Put the Exclusive Merge after A1 and A2 in the model. • Replace the flawed elements (“element’s name”) with correct elements (“element’s name”) before the Exclusive Merge, respectively.

and identifies corresponding element (i.e., activity or a gateway) causing the violation of the LTL rules. The preceding and succeeding elements of that element are matched with the elements of LTL rules to locate the causes of inconsistencies.

The descriptions of the possible causes for each LTL-based primitive and relevant countermeasures to resolve these causes are presented in Table 1. The right-hand side column contains the informal description of elements and second column contains the corresponding LTL-based primitives for formally representing these constructs. Let us consider the first one as an example: sequential order, which describes the relation that one element A2 eventually follows another element A1. As in the other rules in Table 1, violations occur due to a misplacement of elements. For instance, in the case of the sequence described by the LTL rule `(G (A1 -> F A2))`, a violation might happen because the element A2 (transitively) exists in the low-level model as a preceding element of the element A1, but not as a succeeding element of A1. In this context, the counterexample analyzer generates the relevant countermeasures to resolve the violation (in this case: "*swapping the occurrence of A2 to A1*" or "*add A2 after A1*"). Nevertheless, our approach provides promising results for composite controls, for instance, combinations of two or more control structures, like G (ParallelFork -> F (ExclusiveDecision & A1 & .... & An)).

3.2 Visual Support for Understanding and Resolving Inconsistencies

In this section, we explain how the containment checking results can be presented to the developers in a user friendlier manner in comparison to the counterexamples. The visual support aims at shows the developer the causes of containment inconsistencies that occur when the elements and structures (e.g., activities, events and/or gateways) of the high-level process model do not have corresponding parts in the low-level model and also provides relevant countermeasures to resolve the violations.

The visual support is based on the information provided by the counterexample analyzer along with the input low-level process model. In particular, the element(s) that indicates the first element causing the violation of the LTL rule is highlighted in blue whilst the elements that are causes of containment violations are visualized in red, and the elements that satisfied the corresponding LTL rule appear in green. In order to improve the understandability of the counterexamples, we create annotations at the first element causing the violation to show the description of the cause(s) of the containment violation and relevant potential countermeasures to address the violation. Once the root cause of a containment violation is located, the cause is eliminated by updating the involving elements of the low-level process model. To differentiate more than one unsatisfied rule, shades of the particular color are applied, for instance, the first unsatisfied rule is displayed in the original shade while others are gradually represented in lighter tones. The low-level process model displaying highlighted involving elements and annotation of the actual causes of the containment inconsistencies and relevant countermeasures is shown in Figure 3.

4 Use Case from Industrial Case Study

This section briefly discusses a use case from an industrial case study on a billing and provisioning system of a domain name registrar and hosting provider to illustrate the validity of our technique. The Billing Renewal process is taken from our previous industry projects [18]. The Billing Renewal process comprises a wide variety of services, for instance, credit bureau services (cash clearing, credit card validation and payment activities, etc.), hosting services (web and email hosting, cloud hosting, provisioning, etc.), domain services (domain registration, private domain registration, transfer, website forwarding, etc.), and retail services (customer service and support, etc.). Figure 2 shows the high-level Billing Renewal process modeled as a BPMN model. The model is devised to capture essential control structures such as sequence and parallel execution, exclusive decision, and so on. Similarly, the low-level model of the Billing Renewal process containing detailed information is also modeled.

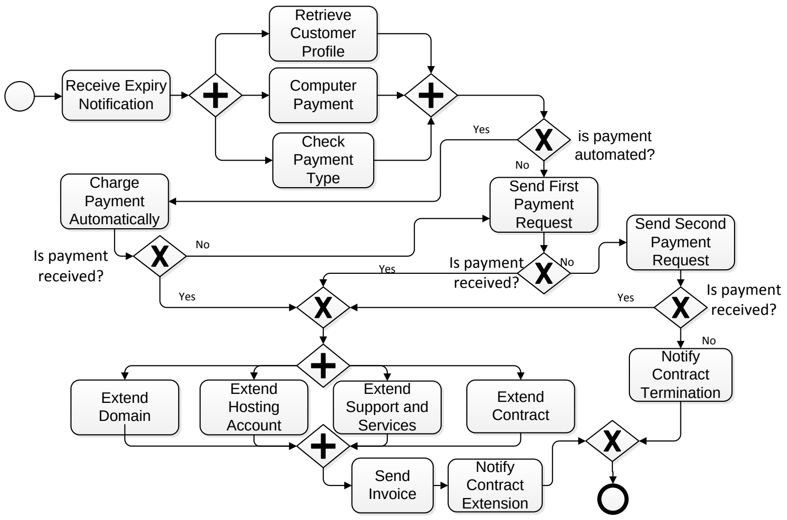
Receive Expiry Notification
Retrieve Customer Profile
Computer Payment
Check Payment Type
is payment automated?
Yes
No
Charge Payment Automatically
Send First Payment Request
Is payment received?
No
Yes
Send Second Payment Request
Is payment received?
Is payment received?
Notify Contract Termination
Extend Domain
Extend Hosting Account
Extend Support and Services
Extend Contract
Send Invoice
Notify Contract Extension

Fig. 2: High-level BPMN model of the Billing Renewal process

Formal consistency constraints (i.e., LTL rules) are automatically generated from the high-level BPMN model whilst the low-level BPMN model is transformed into SMV specification. Next, NuSMV verifies whether the formal SMV specification is consistent with the generated LTL rules. The NuSMV model checker generates a counterexample demonstrating a sequence of permissible state executions leading to a state in which the violation occurs in LTL rule. Finally, our approach for counterexample interpretation is applied to process the violation traces and visualize the involved elements in the low-level BPMN model along with annotations containing containment violation causes and suggestions. We opt to omit the verbose generated LTL rules and SMV specifications and focus more on the interpretation of the generated counterexample.

Listing 1.1 shows an excerpt of a violation trace generated by NuSMV including the list of satisfied and unsatisfied LTL rules, i.e., a counterexample. Despite the size and execution traces of this counterexample, the exact cause of the inconsistency is unclear, for instance, is the containment violation caused by a missing element, or a misplacement of elements, or both of them? It is time consuming and human labor intensive to locate the relevant states because the developers may have to exhaustively walk through all of these execution traces. The counterexample presents symptoms of the cause, but not the cause of the violation itself. Therefore, any manual refinement to the model could fail to resolve the deviation and may introduce other violations.

Figure 3 shows the low-level Billing Renewal process displaying the actual causes of the containment inconsistency and relevant countermeasures to address them. Using the visualizations of the violation, it is easy to see which elements of the low-level model involve in the containment inconsistency. In this case, the containment relationship is not satisfied due to the violation of a parallel fork rule and a sequential rule. The parallel fork rule `G (ParallelFork1 -> F ((ComputerPayment & CheckPaymentType) & RetrieveCustomerProfile))` is unsatisfied because `ParallelFork1` is not followed by the parallel execution of the subsequent tasks (i.e., ComputerPayment, CheckPaymentType and RetrieveCustomerProfile) in the low-level model, which is the actual cause of the containment violation. This violation can be addressed by triggering `ComputerPayment` from `ParallelFork1` as shown in the attached comment to `ParallelFork1`. Similarly, the root cause of second violation is mainly because `SendInvoice` does not lead to `ParallelJoin4`. This might be a symptom of a misplacement of `SendInvoice` in the model as the primary cause that led to the containment inconsistency.

```
$ NuSMV BillingRenweal.smv
...
-- specification G (StartEvent -> F ReceiveExpiryNotification) is true
-- specification G (ReceiveExpiryNotification -> F ParallelFork1) is true
-- specification G (ParallelFork1 -> F ((ComputerPayment & CheckPaymentType) &
 RetrieveCustomerProfile)) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  StartEvent = TRUE
  ReceiveExpiryNotification = FALSE
  ParallelFork1 = FALSE
  RetrieveCustomerProfile = FALSE
  ComputerPayment = FALSE
.....
-- Loop starts here
-> State: 1.6 <-
  CheckPaymentType = FALSE
  ParallelJoin2 = TRUE
  SendLastPaymentRequest = FALSE
  ExclusiveDecision5 = TRUE
  ParallelJoin3 = TRUE
.....
```

Listing 1.1: NuSMV Containment Checking Result of the Billing Renewal Process

The use case illustrates that a rich and concise visualization of the inconsistency causes can allow for an easy identification of the elements that cause the violation and helps developers correct the process model accordingly. In the particular case, after following the suggested countermeasures, rerunning the containment checking process yielded no further violations. Without these supports, the developers would have to study and investigate the syntax and semantics of the trace file in order to determine the relationship between the execution traces and the process model, and then locate the corresponding inconsistency within the model, meaning that the complex matching between the variables and states in the counterexample and the elements of the models must be performed manually.

5 Related Work

The work presented in this paper relates to the two main research areas: behavior model consistency checking and analysis of the model checking results (i.e., counterexamples) for identifying the causes of inconsistencies.

5.1 Behavior Model Consistency Checking

In the literature, many approaches tackle different types of models and/or model checking techniques [12]. However, very few of these studies focus on the consistency of behavior

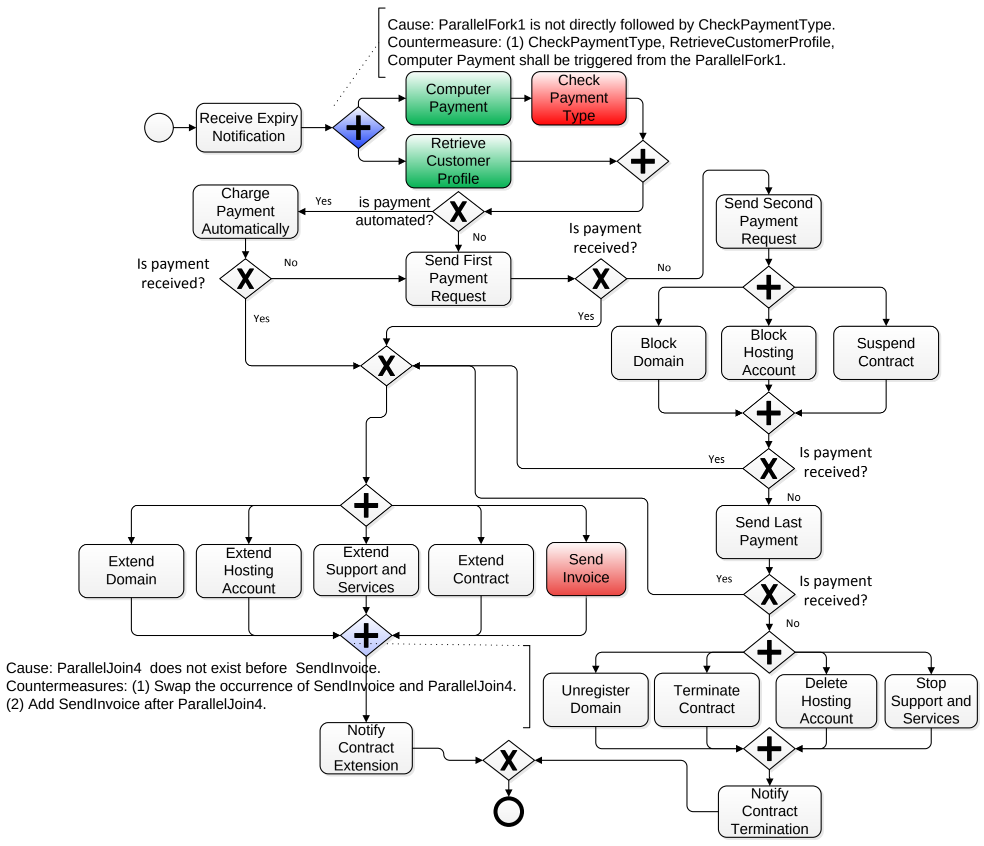
Cause: ParallelFork1 is not directly followed by CheckPaymentType.
Countermeasure: (1) CheckPaymentType, RetrieveCustomerProfile, Computer Payment shall be triggered from the ParallelFork1.
Receive Expiry Notification
Computer Payment
Check Payment Type
Retrieve Customer Profile
Charge Payment Automatically
Yes
is payment automated?
No
Send First Payment Request
Is payment received?
No
Yes
Send Second Payment Request
Block Domain
Block Hosting Account
Suspend Contract
Is payment received?
Yes
No
Send Last Payment
Is payment received?
Yes
No
Extend Domain
Extend Hosting Account
Extend Support and Services
Extend Contract
Send Invoice
Cause: ParallelJoin4 does not exist before SendInvoice.
Countermeasures: (1) Swap the occurrence of SendInvoice and ParallelJoin4.
(2) Add SendInvoice after ParallelJoin4.
Unregister Domain
Terminate Contract
Delete Hosting Account
Stop Support and Services
Notify Contract Extension
Notify Contract Termination

Fig. 3: Visual Support for Understanding and Resolving Containment Violations

models; for instance, van der Straeten et al. [15] present an approach for checking the consistency of different UML models by using description logic. This approach considers model-instance, instance-instance, and model-model conflicts, instead of containment checking. Van der Aalst presents a theoretical framework for defining the semantics of behavior inheritance [1]. In this work, four different inheritance rules, based on hiding and blocking principles, are defined for UML activity diagram, state-chart and sequence diagram. Similar ideas have been presented in [16]. In [10] a general methodology is presented to deal with consistency problem of state-chart inheritance, which involves state-charts as well as the corresponding class diagrams. Communicating Sequential Processes (CSP) is used as a mathematical model for describing the consistency requirements and the FDR tool [2] is used for checking purposes. Weidlich et al. consider the compatibility between referenced process models and the corresponding implementation based on the notion of behavior inheritance [19]. Awad et al. introduce an approach for automated compliance checking of BPMN process models using BPMN-Q queries [3]. They adopted the reduction approach to verify the correctness of process models, instead of performing the detailed analysis using model checker. Unlike our approach, the aforementioned techniques do not aim at providing the analysis of the violation results for identifying the causes of inconsistencies and a

[2] `https://www.cs.ox.ac.uk/projects/fdr/`

set of countermeasures to resolve inconsistencies. Thus, these approaches are very useful for finding similar or alternative behavioral descriptions but not applicable for verifying the containment relationship.

5.2 Generating and Analyzing Counterexamples

The problem of generating and analyzing model-checking counterexamples are classified as follows: generating the counterexample efficiently, automatically analyzing the counterexample to extract the exact cause of violations, and creating a visualization framework suitable for interactive exploration.

Several existing approaches have addressed the idea of generating proofs from the model-checking runs. Many of these techniques focus on building evidence in form of a proof and controlling the generation of information to aid the understanding of counterexamples [7, 17]. One of the drawbacks of these approaches is their size and complexity, which can be polynomial in the number of states of the system and of exponential length in the worst case. The proof-like witness techniques also require manual extrapolation, and the developers still need certain knowledge of the underlying formalisms in order to understand the proofs.

The problem of the automated analysis of counterexamples was addressed by many researchers, for instance, Ball et al. [4] describe an error trace as a symptom of the error and identify the cause of the error as the set of transitions in an error trace that does not appear in a correct trace of the program via the SLAM model checker. Kumazawa and Tamai present an error localization technique LLL-S for a given behavior model. The proposed technique identifies the infinite and lasso-shaped witnesses that resemble the given counterexample [11]. However, these approaches focus on finding the error causes in the program, such as deadlocks, assertion violations, and so on, but they are not applicable for verifying the causes of unsatisfied containment relationships.

Visual presentation of generated counterexamples is explored by Dong et al. [9]. The authors developed a tool that simplifies the counterexample exploration by presenting evidence for modal μ-calculus through various graphical views. In particular, the highlighting correspondence between the generated counterexample and the analyzed property is addressed in their visualization process. Armas-Cervantes et al. [2] developed a tool for identifying behavioral differences between pairs of business process models by using Asymmetric Event Structure (AES) and verbalization of the results.

The above discussed approaches focus on general consistency checking. In contrast to our work, none of these techniques focuses on the diagnosis of counterexamples generated by a model checker with respect to containment checking. Note that, in our approach we do not need nor modify the source code of the model checker. Our interpretation process automatically extracts the information from the generated counterexample. In addition, another differentiating factor of our approach in comparison to aforementioned approaches is that our framework provides a compact and concise representation of the failure causes (such as missing or misplacement of elements) and countermeasures that are easily understandable for non-expert stakeholders. Finally, the counterexample interpretation steps are fully automated and do not require developer intervention.

6 Conclusion and Future Work

In this paper, we presented an approach for interpreting the causes of inconsistencies in behavior models based on the output results (i.e., counterexamples) of the model checker. In our work, the counterexample analyzer will help locating the actual causes of a containment inconsistency and producing appropriate guidelines as countermeasures based on the information extracted from counterexample trace file, formalization rules (i.e., LTL-based primitives), and the SMV specification of the low-level models. The visual support will show the involving elements along with annotations of causes and countermeasures in the low-level model. The advantage of this interpretation technique is twofold. On the one hand, the technique supports users who have limited knowledge of the underlying formalisms, and therefore, are not proficient in analyzing the cryptic and verbose counterexamples. On other hand, by locating actual cause(s) of the inconsistency and providing the relevant countermeasures to alleviate the inconsistencies to the user, it significantly reduces the time of manually locating the causes of an inconsistency. To the best of our knowledge, we investigated and presented almost all possible causes of a containment inconsistency and relevant countermeasures represented by a specific LTL-based primitive.

Currently, our approach supports output generated by the NuSMV model checker with respect to containment checking. Nevertheless, it is possible to adapt the presented techniques to support the other behavior models such as state machines, sequence diagrams[3], and BPEL[4] with reasonable extra efforts. In our future work, we plan on extending our approach for additional constructs (rather than the set of essential widely used constructs presented in this paper) and evaluate our approach with larger case studies. Another direction for future work is to quantitatively evaluate the counterexample interpretation approach in order to explore the pragmatic usability of the approach. Finally, we are looking into ways to extend our approach, in particular to support the interpretation and visualization of the analysis results from model checkers such as SPIN[5] that employ different underlying model checking techniques and data structures. The counterexample analyzer depends on the formalization rules and the options used in the model checker for generating output trace file. Extending our approach for SPIN is also possible as NuSMV and SPIN share several similar concepts regarding the state based counterexamples.

Acknowledgment. The research leading to these results has received funding from the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001.

References

[1] Van der Aalst, W.M.: Inheritance of dynamic behaviour in uml. MOCA 2, 105–120 (2002)

[2] Armas-Cervantes, A., Baldan, P., Dumas, M., García-Bañuelos, L.: Behavioral comparison of process models based on canonically reduced event structures. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) Business Process Management, Lecture Notes in Computer Science, vol. 8659, pp. 267–282. Springer International Publishing (2014)

[3] Awad, A., Decker, G., Weske, M.: Efficient compliance checking using bpmn-q and temporal logic. In: Proceedings of the 6th International Conference on Business Process Management. pp. 326–341. BPM '08, Springer-Verlag, Berlin, Heidelberg (2008)

[4] Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 97–105. POPL '03, ACM, New Orleans, Louisiana, USA (2003)

[3] `http://www.omg.org/spec/UML/2.4.1`

[4] `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`

[5] `http://spinroot.com/spin/whatispin.html`

[5] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: a new symbolic model checker. International Journal on Software Tools for Technology Transfer 2(4), 410–425 (2000)

[6] Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: 11th Int'l Conf. on Computer Aided Verification (CAV). pp. 495–499. Springer-Verlag, London, UK (1999)

[7] Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference. pp. 427–432. DAC '95, ACM, New York, NY, USA (1995)

[8] Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)

[9] Dong, Y., Ramakrishnan, C.R., Smolka, S.: Model checking and evidence exploration. In: Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the. pp. 214–223 (April 2003)

[10] Engels, G., Heckel, R., Küster, J.M.: Rule-based specification of behavioral consistency based on the uml meta-model. In: 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. pp. 272–286. Springer-Verlag, London, UK, UK (2001)

[11] Kumazawa, T., Tamai, T.: Counter example-based error localization of behavior models. In: Proceedings of the Third International Conference on NASA Formal Methods. pp. 222–236. NFM'11, Springer-Verlag, Berlin, Heidelberg (2011)

[12] Lucas, F.J., Molina, F., Toval, A.: A systematic review of UML model consistency management. Information and Software Technology 51(12), 1631–1645 (Dec 2009)

[13] Muram, F.U., Tran, H., Zdun, U.: Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking. In: 11th Int'l Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA). pp. 93–107. Grenoble, France (2014)

[14] Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. SFCS '77, IEEE Computer Society, Washington, DC, USA (1977)

[15] van der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between uml models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003 – The Unified Modeling Language. pp. 326–340. Springer, Berlin, Heidelberg (2003)

[16] Stumptner, M., Schrefl, M.: Behavior consistent inheritance in uml. In: Laender, A., Liddle, S., Storey, V. (eds.) Conceptual Modeling ER 2000, Lecture Notes in Computer Science, vol. 1920, pp. 527–542. Springer-Verlag, Berlin, Heidelberg (2000)

[17] Tan, L., Cleaveland, R.: Evidence-based model checking. In: Brinksma, E., Larsen, K. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 2404, pp. 455–470. Springer Berlin Heidelberg (2002)

[18] Tran, H., Zdun, U., Dustdar, S.: Name-based view integration for enhancing the reusability in process-driven soas. In: zur Muehlen, M., Su, J. (eds.) Business Process Management Workshops, Lecture Notes in Business Information Processing, vol. 66, pp. 338–349. Springer Berlin Heidelberg (2011)

[19] Weidlich, M., Dijkman, R., Weske, M.: Behaviour Equivalence and Compatibility of Business Process Models with Complex Correspondences. The Computer Journal 55(11), 1398–1418 (Feb 2012)