A Testing Approach for Hidden Concurrencies based on Process Execution Logs

Kristof Böhmer and Stefanie Rinderle-Ma

University of Vienna, Faculty of Computer Science, Vienna, Austria
{kristof.boehmer,stefanie.rinderle-ma}@univie.ac.at

Abstract. It is crucial to ensure correct process model executions. However, existing process testing approaches struggle with the verification of concurrent resource access patters that can lead to concurrency faults, such as, deadlocks or data corruption during runtime. Thus, we provide a concurrency verification approach that exploits recorded executions to verify the most frequently occurring concurrent resource access patterns with low test execution time. A prototypical implementation along with real life and artificial process execution logs is utilized for an evaluation.

Keywords: Process Testing, Concurrency, Test Case Prioritization

1 Introduction

Ensuring fault free process executions is crucial [8]. However, this becomes challenging due to the increased complexity and interconnectivity of processes and their invoked services and applications [7] (i.e., shared resources). Moreover, organizations utilize a huge amount of process models [6], where each model likely spawns multiple *concurrently executed* process instances [13].

This can result in *Hidden Concurrencies* (HC). HCs are caused by concurrent activity executions which invoke the same *shared resources*. Figure 1 depicts an example HC. Two instances I1 and I2 are executed concurrently on two process models P1 and P2 (abstract notation inspired by Petri Nets). I1 and I2 both access the same *shared resources*, i.e., services S1 and S2, through activity execution. P2 also contains an obvious (modeled) concurrency based on a parallel split. The HC is caused by the concurrent access to service S2 by I1 and I2, i.e., activities in I1 and I2 invoke S2 within an overlapping time span.

The hidden concurrent access to S2 by I1 and I2 does not become evident at design time and might lead to a concurrency fault at runtime (i.e., a HC fault) *iff* S2 struggles when dealing with the access patterns caused by I1 and I2 (i.e., each HC fault is related to a HC, however, a HC can but does not necessarily lead to a HC fault). A *HC fault* occurs if multiple activities concurrently access the same shared resource in a way that creates inconsistencies/deadlocks [14]. Note, that developers do not always employ synchronization [2], paving the way for HC faults when a resource is concurrently accessed by multiple instances.

Process instances are isolated from each other. However, their access on potentially faulty shared resources is *not* isolated which can result in HC faults.

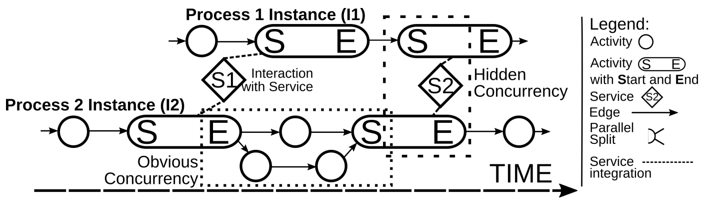

Fig. 1. *Hidden* and *obvious* (modeled) concurrent executions - example

Moreover it is not obvious if/how multiple resources interact with each other. These interactions can lead to HC faults even if two "different" resources are accessed during concurrent executions. Moreover we found that current research on process verification, cf. [4], has neglected the detection of HCs so far.

Overall this paper addresses the following research questions:

RQ1 How can HCs in process instance executions be detected?
RQ2 How can the most likely HCs be verified with low testing efforts?
RQ3 How can the efficiency of the presented approaches be evaluated?

In order to address above research questions, an automatic verification heuristic for HCs is proposed. It exploits recorded process executions to determine the most frequently occurring HCs $\mapsto$ **RQ1**. Subsequently this information is utilized to select process test cases which verify the most frequently ocurring concurrent resource access patterns $\mapsto$ **RQ2**. The conducted evaluation shows the efficiency and applicability of the presented approach $\mapsto$ **RQ3**.

This paper is organized as follows. Approaches to identify and prevent potential process model execution HC faults are discussed in Section 2. Evaluation, corresponding results, and their discussion are presented in Section 3. Section 4 discusses related work. Conclusions and future work is given in Section 5.

2 Preventing Hidden Concurrency Faults

Let $\mathcal{U}$ denote a *process repository* containing units u. Note that each unit $u \in \mathcal{U}$ is unique and represents a single activity. Assume further that *executions* of $u \in \mathcal{U}$ are logged. A unit's execution is reflected by an execution event $e := (u, t_s, t_e)$, where t_s and t_e denote time stamps which reflect the start and end of u's execution. Finally, let a bag O hold all execution events e for a given process repository $\mathcal{U}$. O can be interpreted as a simple execution log for process activities and instances, i.e., a collection of execution events over a process repository.

Let a test case, in short test, $t \in \mathcal{T}$, consist of a set of process model elements which are verified/covered by t [3]. Testing units $u \in \mathcal{U}$ is assumed sufficient, hence, $t :\subseteq \mathcal{U}$. Complete test definitions include additional information (e.g., expected variables values) which enable to detect concurrency faults.

Identifying Hidden Concurrencies HCs are identified by analyzing the executions of each process unit. Those executions can be extracted from process execution logs which are generated by process execution engines. The presented approach identifies HCs based on *unit pairs*, $(u, u'), u, u' \in \mathcal{U}$, i.e., a pair of activities. Unit pairs are utilized because *a*) a unit pair is the smallest entity that can provoke a HC; and is *b*) concentrating on a minimal amount of interactions at once to simplify the interpretation of the findings; and *c*) complex concurrency fault conditions can be represented by grouping units and unit pairs.

Basically, HCs can be observed in four flavors, cf. [1], depending on how the concurrent execution of two units $u, u' \in \mathcal{U}$ overlaps, cf. Def. 1:

Definition 1 (Overlapping Flavors). *Let $\mathcal{U}$ be a set of units and O be the bag of associated unit executions. Let further $e_1 = (u, t_{s_1}, t_{e_1})$, $e_2 = (u', t_{s_2}, t_{e_2}) \in O$ be two executions for a unit pair $u, u' \in \mathcal{U}$ and let $ov \in [0;1]$ be an overlapping factor. e_1, e_2 can be related under the following overlapping flavors:*

$$OvlpF := O \times O \times [0;1] \mapsto \{\textit{start/end, complete, almost, no}\}$$

$$OvlpF(e_1, e_2, ov) =$$

$$:= \begin{cases} \textit{start/end} & \textit{iff } (t_{s_2} \leq t_{s_1} \wedge t_{e_2} \geq t_{s_1} \wedge t_{e_2} < t_{e_1}) \vee (t_{s_2} \geq t_{s_1} \wedge t_{s_2} \leq t_{e_1} \wedge t_{e_2} > t_{e_1}) \\ \textit{complete} & \textit{iff } (t_{s_1} \leq t_{s_2} \wedge t_{e_1} \geq t_{e_2}) \vee (t_{s_2} \leq t_{s_1} \wedge t_{e_2} \geq t_{e_1}) \\ \textit{almost} & \textit{iff } (t_{e_2} < t_{s_1} \vee t_{e_1} < t_{s_2}) \wedge (t'_{e_2} > t'_{s_1} \vee t'_{e_1} > t'_{s_2}) \\ \textit{no} & \textit{otherwise} \end{cases}$$

where

$$d_1 := t_{e_1} - t_{s_1},\ d_2 := t_{e_2} - t_{s_2},$$
$$t'_{s_1} := t_{s_1} - d_1 \cdot ov, t'_{e_1} := t_{e_1} + d_1 \cdot ov,$$
$$t'_{s_2} := t_{s_2} - d_2 \cdot ov, t'_{e_2} := t_{e_2} + d_2 \cdot ov$$

Definition 1 is used to identify HCs for unit pairs in an execution log O. For this Eq. 1 compares the executions of all unit pairs $u, u' \in \mathcal{U}$ and determines the respective overlapping flavors and, hereby, the associated *HC risk*. The HC risk expresses how likely a HC can be observed for a given unit pair. Note, if a HC is observed frequently (high HC risk), then a related concurrency fault can have a high impact on process execution correctness, cf. [14].

$$Ovlp(O, ov) = \{(e.u, e', f) | e, e' \in O \wedge e.u \neq e'.u \wedge f := \text{OvlpF}(e, e', ov) \wedge f \neq no\} \quad (1)$$

Def. 2 calculates the HC risk of two units $u, u' \in \mathcal{U}$. Note, that the the *min* function limits the concurrent execution likelihood of u and u' (i.e., the HC risk) to an interval of $[0, 1]$. Otherwise the HC risk could exceed > 1 if u' would be concurrently executed more than once for each execution of u.

Definition 2 (Concurrency Risk). *Let $\mathcal{U}$ be a set of units and O be the bag of associated unit executions. Let further $Ovlp(O, ov)$ be a set of overlapping units, cf. Eq. 1. Then the HC risk for two units $u, u' \in U$ is calculated as*

$$ConRisk(u, u') = min(\frac{co \cdot ct + seo \cdot set + ao \cdot at}{te}, 1)$$

where

- $te := |\{e \in O | e.u = u\}|$;
- $ovrlppngExectns := \{o := (u, e, f) \in Ovlp(O, ov) | o.u = u \wedge o.e.u = u'\}$;
- $ce := |ovrlppngExectns|$;
- $co := |\{(u, e, f) \in ovrlppngExectns | f = complete\}|$;
- $ao := |\{(u, e, f) \in ovrlppngExectns | f = almost\}|$;
- $seo := |\{(u, e, f) \in ovrlppngExectns | f = start/end\}|$;

– $ct, set, at \in [0,1]$ *weigh the different overlapping flavors (tuning variables)*

Def. 2 considers the number of executions of u (te), the number of executions of u that overlap with an execution of u' (ce), as well as the number of almost (ao), start/end (seo), and completely overlapping (co) executions of u with u'. Moreover, the executions can be weighed along the overlapping flavors using *tuning variables* (ct, set, at). This enables to model, for example, that an almost overlapping execution only represents a likely HC, i.e., it should not have the same impact on the calculated HC risk as, for example, a complete overlapping.

Eq. 2 calculates the HC risk for all unit pairs $u, u' \in U \subseteq \mathcal{U}$:

$$OvlpRisk(U) = \{(u, u', r) | u, u' \in U \wedge u \neq u' \wedge r := \text{ConRisk}(u, u')\} \qquad (2)$$

Calculating the HC risk for all unit pairs $u, u' \in U \subseteq \mathcal{U}$ enables the identification of units that frequently experience hidden concurrent executions. This is exploited to select a set of test cases which reaches a high amount of verified hidden concurrencies with a low amount of test cases and testing effort.

Test Group Selection Groups of test cases are applied to identify HC faults. This is because the verification of HCs requires that multiple instances are executed concurrently. However, each test only spawns and verifies the execution of a single instance. Hence, for each unit u multiple test cases must be combined to a test group which verifies the HCs of u. So, it is necessary to identify which test cases should be selected from the existing set of test cases $\mathcal{T}$ and combined to test groups $tg :\subseteq \mathcal{T}$. As concurrent executions are the precondition for HC faults, the proposed approach relies on the HC risk during test group construction.

Intuitively, for unit u all units u' that have a non-zero HC risk with u, i.e., $\exists(u, u', risk) \in OvlpRisk(\mathcal{U})$, could be considered. However, this can result in large testing efforts or the non-verification of resource access patters which require the interaction of several units/instances. Alg. 1 addresses these considerations by restricting the number of considered u' with $\exists(u, u', risk) \in OvlpRisk(\mathcal{U})$ to the top $gDist$ ones ($gDist \in \mathbb{N}$) with respect to HC risk. Def. 3 provides a function for determining units with maximum HC risk.

Definition 3 (Projection on unit with maximum risk). *Let $OvlpRisk(\mathcal{U})$ be the set of overlapping units for the set of all units $\mathcal{U}$. For a given unit $u \in \mathcal{U}$, function maxRisk determines unit $u' \in \mathcal{U}$ which is the unit with the maximum HC risk in $OvlpRisk(\mathcal{U})$. Formally:*

$maxRisk : \mathcal{U} \times 2^{OvlpRisk(\mathcal{U})} \mapsto \mathcal{U}$
$maxRisk(u, OvlpRisk(\mathcal{U})) = u'$
  with $\exists(u, u', r) \in OvlpRisk(\mathcal{U}) \wedge r = max\{r' \mid \exists o \in RelU \text{ with } o.r = r'\}$
where $RelU := \{o \in OvlpRisk(\mathcal{U}) \mid o.u = u)\}$

Alg. 1 creates an independent unit group for each unit $u \in \mathcal{U}$ considering its HC risk with concurrently executed $u' \in \mathcal{U}$ and the HC risks of transitively related units $u'' \in \mathcal{U}$, i.e., units that are executed concurrently with u'.

We can illustrate a unit group, as determined by Alg. 1, as a tree structure which uses u as its root note. Fig. 2 depicts the construction of a unit group for

```
Algorithm DetUGrp($u$, $OvlpRisk(U)$, $gDist$, $gLvl$)
    Data: $u \in U$, $OvlpRisk(U)$, $gDist$, $gLvl$
    Result: set $UGroup(u)$ for $u$
    $UGroup(u) := \{u\}$
    for $i=0; i < gDist \wedge gLvl \geq 0; i{+}{+}$ do
        $maxR := maxRisk(u, OvlpRisk(U))$ acc. to Def. 3
        $OvlpRisk(U) := OvlpRisk(U) \setminus \{o \in OvlpRisk(U) \mid o.u = u, o.u' = maxR\}$
        $UGroup(u) := UGroup(u) \cup DetUGrp(maxR, OvlpRisk(U), gDist, gLvl - 1)$
    return $UGroup(u)$
```

Algorithm 1: Construct unit group for unit u based on the HC risk

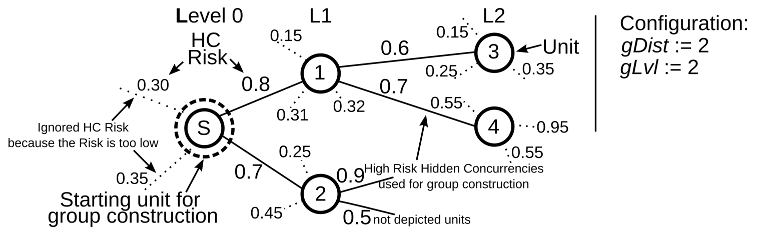

Fig. 2. Construction of a unit group, starting from unit S

unit S. By analyzing the HC risk of S it is detected that unit ① and ② have the highest risk to be executed concurrently with S. Hence, in a first step the unit group of S collects these two units ($gDist = 2$). Analogously, the search for related units with the highest HC risk is expanded to ①. The search stops at unit ③ and ④ as the maximum unit group level ($gLvl = 2$) is reached. Note, the same expansion is applied on unit ② (not depicted).

Subsequently, each unit group $UGroup(u)$ is transformed into a new test group $TGroup(t)$. For each unit u in the analyzed unit group a test case $t \in \mathcal{T}$, were u is covered by t, is chosen and added to a test group, cf. Alg. 2. This step is repeated until for each $u \in UGroup(u)$ a $t \in \mathcal{T}$ was added to $TGroup(t)$ that verifies the correctness of u. In Alg. 2 $randSelect(u, \mathcal{T}) = t$ with t covers u.

```
Algorithm DetTestGroup($\mathcal{T}$, $UGroup(u)$)
    Data: all tests $\mathcal{T}$ and a unit group $UGroup(u)$
    Result: a test group $TGroup(t)$
    $TGroup(t) := \emptyset$
    foreach $u \in UGroup(u)$ do
        $test$:=randSelect({ $t \in \mathcal{T}$ | u is covered by t})
        $TGroup(t) := TGroup(t) \cup \{test\}$
    return $TGroup(t)$
```

Algorithm 2: Transforming a unit group into a test group

Test Group Prioritization As an individual test group is created for each unit, executing each test group can take a substantial amount of time. Typically this problem is tackled by test case prioritization, cf. [9]. However, existing test case prioritization techniques are not applicable for the presented approach, cf. [4], because they *a*) only rank single test cases (i.e., test case groups are not supported); and *b*) are not specifically tailored for hidden concurrency testing.

Hence, this paper proposes a novel prioritization approach which uses seven metrics that focus on hidden concurrency fault detection. The **primary prioritization metrics** P are: test group execution time, test diversity, and HC risk. In addition, the **secondary prioritization metrics** S are: amount of test cases, covered back-end systems, additional coverage, and multi unit coverage.

The $PrioValue(tg, P, S)$ of a test group tg is determined based on Eq. 3:

$$PrioValue(tg, P, S) := \sum_{i=0}^{|P|} \frac{1}{|P|} \cdot Prio_i^P(tg) + \sum_{j=0}^{|S|} \frac{1}{2 \cdot |S|} \cdot Prio_j^S(tg) \tag{3}$$

$Prio_i^P(tg)/Prio_i^S(tg)$, cf. Eq. 3, denote functions that determine the normalized primary/secondary metrics from P/S as described above. Note, that the division by 2 for secondary metrics decreases their influence. Eq. 3 enables to repeatedly identify and subsequently execute the tg with the maximum identified $PrioValue(tg, P, S)$ until all test groups are executed.

3 Evaluation

The evaluation test data consists of *real life process execution logs* (BPIC) from the BPI Challenge 2015[1] and *artificial logs* (TeleClaim) which describe the handling of insurance claims in call centers (source: [12][2]).

The real life log data consists of 262,628 events, 5,649 process instance execution paths, and 398 activities – recorded from 2010 to 2015 and provided by five Dutch municipalities (BPIC15_1 to BPIC15_5). The artificial log data consists of 46,138 events, 3,512 process instance execution paths, and 11 unique activities. All evaluated logs contain the start and end time of each activity execution.

Metrics and Evaluation The evaluation was designed to assess if HCs occur during process model executions. Subsequently, it was checked if existing load testing approach are sufficient to test identified HCs. Finally, the efficiency of the proposed test prioritization approach is evaluated. For this multiple prioritization approaches are compared using the Average Percentage of Faults Detected (APFD) metric, cf. [9]. A high APFD ensures a high fault detection rate with a minimal amount of test group executions and test group execution time.

The APFD $\in [0, 1]$ is calculated using Eq. 4. Hereby, n is the number of test groups TG, m is the number of known faults F to search for, and $Pos(tg, F_i)$ identifies the rank/position of the test group $tg \in TG$ that identifies fault $F_i \in F$.

$$APFD(n, m, TG, F) = 1 - \frac{\sum_{i=0}^{m} Pos(tg, F_i)}{n \cdot m} + \frac{1}{2 \cdot n} \tag{4}$$

Note, the random aspects of the evaluation were evened out by executing it 100 times and taking the average result. The HC faults, which are "searched", in the following, were generated by randomly selecting pairs of activities that are executed concurrently, i.e., a HC. These randomly chosen activities are then marked as faulty and the analyzed testing approaches strive to construct test groups which cover those faulty HCs (11/3 HCs were marked as faulty for BPIC/TeleClaim). Each recorded execution path is covered by at least one test.

[1] `http://www.win.tue.nl/bpi/2015/challenge`—DOI: 10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1

[2] `http://www.processmining.org/event_logs_and_models_used_in_book`

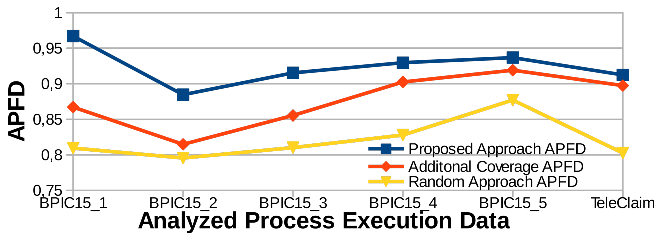

Fig. 3. Efficiency of the presented test group prioritization approach

Results The results were generated by analyzing the BPIC and TeleClaim execution log files with a proof-of-concept implementation of the presented approach. The JAVA 7 source code/documentation of the implementation can be found on GitHub at `https://github.com/KristofGit/Hidden_Concurrency`.

The evaluation utilized an overlapping factor ov of 0.1 while the HC risk tuning variables were set to $at = 0.1$, $set = 0.8$, and $ct = 1$. The BPIC ($gDist = 5/gLvl = 3$) and TeleClaim ($gDist = 2/gLvl = 2$) data was analyzed with different values for $gDist/gLvl$ because the TeleClaim processes are simpler than the BPIC processes so that smaller test groups are sufficient.

We found that the analyzed execution logs contained tens of thousands HCs for each overlapping flavor. Hence, we checked if existing load testing based approaches are sufficient to test each HC in a reasonable amount of test execution time. Unfortunately, we found that load testing would require about 27 days test execution time to identify all HC faults in the TeleClaim processes. Note, the BPIC processes are more complex and would require even more test execution time. Hence, test prioritization techniques are a necessity.

The evaluation shows (cf., Eq. 4 and Fig. 3) that the proposed approach creates the fastest test group ranking/execution order which identifies all artificial HC faults in the least amount of time. Note, that the proposed approach not only creates a better result than the baseline random approach but also as the additional coverage based approach which is a standard approach in existing work, cf. Fig. 3. When applying test group prioritization the amount of test group executions, required to identify all faults, is reduced to 127 (BPIC) and 2 (TeleClaim) because only minimal set of all available test groups must be executed to identify all faults. This significantly reduces the test group execution time, i.e., about 52 minutes would be required to identify all TeleClaim HC faults.

4 Related Work

Support for concurrency fault detection in the business process domain is limited, cf. [4]. For example, existing process testing approaches (cf. [11,15,10,5]) ignore concurrency or only consider a single process so that HCs will, most likely, not be identified. The last drawback applies to all the found work, cf. [4].

The most advanced approach, [5], reduces the concurrency testing workload by incorporating back-end services during test case selection by focusing on activities which could concurrently access the same back-end service. However, this work still ignores hidden concurrency faults generated by the concurrent execution of multiple processes and process instances (i.e., it only concentrates on a single process model and obvious modeled concurrent control flow paths).

5 Conclusions

The proposed test prioritization heuristic reduces the testing effort to process model execution scenarios and units which most likely trigger concurrency faults $\mapsto$ **RQ1** and **RQ2**. Moreover, the proposed algorithms were designed in a configurable fashion so that also very rarely occurring concurrency faults can be identified. The evaluation results show the efficiency/applicability of the presented approach for real life and artificial processes $\mapsto$ **RQ3**.

Acknowledgment This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-072.

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. ACM 26, 832–843 (1983)
2. Alrifai, M., Dolog, P., Nejdl, W.: Transactions concurrency control in web service environment. In: Web Services. pp. 109–118. IEEE (2006)
3. Böhmer, K., Rinderle-Ma, S.: A genetic algorithm for automatic business process test case selection. In: On the Move: CoopIS. pp. 166–184. Springer (2015)
4. Böhmer, K., Rinderle-Ma, S.: A systematic literature review on process model testing: Approaches, challenges, and research directions. arXiv (2015)
5. De Angelis, F., Fanì, D., Polini, A.: Partes: A test generation strategy for choreography participants. In: Automation of Software Test. pp. 26–32. IEEE (2013)
6. Dijkman, R.M., Rosa, M.L., Reijers, H.A.: Managing large collections of business process models - current techniques and challenges. Computers in Industry 63(2), 91–97 (2012)
7. Fdhila, W., Rinderle-Ma, S., Indiono, C.: Change propagation analysis and prediction in process choreographies. Cooperative Information Systems 24, 47–62 (2015)
8. Leymann, F., Roller, D.: Production workflow concepts and techniques. Prentice Hall PTR (2000)
9. Malishevsky, A.G., Ruthruff, J.R., Rothermel, G., Elbaum, S.: Cost-cognizant test case prioritization. Tech. rep., University of Nebraska-Lincoln (2006)
10. Ruth, M.E.: Concurrency in a decentralized automatic regression test selection framework for web services. In: Mardi Gras Conference. pp. 7:1–7:8. ACM (2008)
11. Sriganesh, S., Ramanathan, C.: Externalizing business rules from business processes for model based testing. In: Industrial Technology. pp. 312–318. IEEE (2012)
12. Van Der Aalst, W.: Process mining: discovery, conformance and enhancement of business processes. Springer (2011)
13. Van Der Aalst, W.M., Ter Hofstede, A.H., Weske, M.: Business process management: A survey. In: Business process management, pp. 1–12. Springer (2003)
14. Wang, C., Said, M., Gupta, A.: Coverage guided systematic concurrency testing. In: Conference on Software Engineering. pp. 221–230. ACM (2011)
15. Yuan, Y., Li, Z.J., Sun, W.: A Graph-Search Based Approach to BPEL4WS Test Generation. In: Software Engineering Advances. p. 14. IEEE (2006)