Mining Sequential Patterns to Explain Concurrent Counterexamples*

Stefan Leue[1] and Mitra Tabaei Befrouei[2]

[1] University of Konstanz
Stefan.Leue@uni-konstanz.de
[2] Vienna University of Technology
Tabaei@forsyte.at

Abstract. Concurrent systems are often modeled using an interleaving semantics. Since system designers tend to think sequentially, it is highly probable that they do not foresee some interleavings that their model encompasses. As a consequence, one of the main sources of failure in concurrent systems is unforeseen interleavings. In this paper, we devise an automated method for revealing unforeseen interleavings in the form of sequences of actions derived from counterexamples obtained by explicit state model checking. In order to extract such sequences we use a data mining technique called *sequential pattern mining*. Our method is based on contrasting the patterns of a set of counterexamples with the patterns of a set of correct traces that do not violate a desired property. We first argue that mining sequential patterns from the dataset of counterexamples fails due to the inherent complexity of the problem. We then propose a reduction technique designed to reduce the length of the execution traces in order to make the problem more tractable. We finally demonstrate the effectiveness of our approach by applying it to a number of sample case studies.

Keywords: concurrency bugs, counterexample explanation, sequential pattern mining, model checking.

1 Introduction

Concurrency bugs are among the most difficult software bugs to detect and diagnose. This is mainly due to the inherent inability of humans to comprehend concurrently executing computations and to foresee the possible interleavings that they can entail. The interleaving semantics commonly used to interpret the computation of concurrent systems imposes a total order on the execution of concurrent actions in a system. Concurrency is then interpreted as non-deterministic choices between different interleavings. System designers are used to thinking

* Supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF), by Deutsche Forschungsgemeinschaft (DFG) through the grants "DiRePro" and "IMCOS" and by the Vienna Science and Technology Fund (WWTF) through grants VRG11-005, PROSEED, and ICT12-059.

E. Bartocci and C.R. Ramakrishnan (Eds.): SPIN 2013, LNCS 7976, pp. 264–281, 2013.

sequentially when designing the model of a system. In concurrent systems it is therefore highly probable that they do not foresee some interleavings that their model encompasses. It is therefore a widely held view that one of the main sources of failure in concurrent systems is unforeseen interleavings resulting in undesired system behavior.

Model checkers are particularly well-suited for detecting concurrency bugs due to the exhaustive exploration of all possible interleavings of the concurrent actions that they perform. They can therefore reveal bugs which are impossible or difficult to find by testing methods. However, counterexamples generated by model checking tools only indicate symptoms of faults in a model, they do not offer aid in locating faults in the code of the model. In order to locate a root cause for a counterexample in the code of a model a significant amount of manual analysis is required. Since the manual inspection of lengthy counterexamples of sometimes up to thousands of events is time consuming and error prone, an automatic method for explaining counterexamples that assists model designers in localizing faults in their models is highly desirable.

In this paper we aim at developing an automated method for explaining counterexamples indicating the violation of a desired property in concurrent systems. Our method benefits from the analysis of a large number of counterexamples that can be generated by a model checking tool such as SPIN [9]. We refer to the set of counterexamples that show how the model violates a given property as the *bad* dataset. With the aid of SPIN, it is also possible to produce a set of execution traces that do not violate the desired property. We refer to this set of non-violating traces as the *good* dataset.

For explaining counterexamples, we examine the differences in the traces of the good and bad datasets, which is the foundation of a large number of approaches for locating faults in program code (see, for instance, [27]). Lewis' theory of causality and counterfactual reasoning provides justification for this type of fault localization approaches [13].

To reveal unforeseen interleavings in the form of sequences of actions, we use a data mining technique called *sequential pattern mining* or *frequent subsequence mining* [1,4]. This data mining technique has diverse applications in areas such as the analysis of customer purchase behavior, the mining of web access patterns and the mining of motifs in DNA sequences. Frequent subsequence mining is an active area of research and a number of algorithms for mining frequent subsequences have been developed which have been proven to be efficient in practice with respect to various test datasets [26,25,20].

By contrasting the sequential patterns of the good and bad datasets, we extract a set of sequences of actions that are only common in the bad dataset but not common in the good dataset. We refer to this approach as *contrast mining* and to the resulting patterns as *anomalies*. We assume that these anomalies can reveal to the model designer unforeseen interleavings or unexpected sequences of actions that cause the violation of a desired property.

The contributions of this work are as follows:

1. We propose an automated method based on contrast mining for explaining concurrency bugs.
2. We propose a length reduction technique to make the mining problem more tractable.
3. We show how concurrency bugs can be explained in general by only analyzing the good and the bad traces and without exploiting the characteristics of specific bugs such as data races or atomicity violations.

In our precursory work on explaining counterexamples [12] we extract ordered sequences of events consisting of contiguous events inside counterexamples. In this work, we improve our explanation by extracting sequences of events which do not necessarily occur contiguously inside counterexamples.

Structure of the Paper. Section 2 gives the definition of the problem and also motivates the problem by introducing a running example. Section 3 describes in detail our proposed method for explaining counterexamples. We then present experimental results in Section 4. Section 5 discusses closely related work from different domains. Section 6 concludes with a note on future work.

2 Problem Definition

2.1 Basic Concepts

Our goal is to identify ordered sequences of non-contiguous events that explain the violation of a safety property in a concurrent system. Such a violation represents that there exist undesired or unsafe states which are reachable by system executions. We use the explicit state model checker SPIN [9] in order to compute system executions represented as sequences of events that lead from an initial state of the system into a property violating state, often referred to as counterexamples. We use linear temporal logic (LTL) [2] to specify properties and we use $\sigma \not\models \varphi$ to express that a counterexample σ violates an LTL property φ.

Definition 1. *Let* Act *denote the finite set of actions in a concurrent system. If counterexample σ violates the safety property φ, then σ will be a finite sequence of events denoted as $\langle e_1, e_2, ..., e_n\rangle$ where each e_i corresponds to the execution of an action in the system.*

In fact, the finite set of actions, *Act*, corresponds to the Promela statements [9] of the concurrent system models verified by the SPIN model checker. According to Def. 1, we may use the terms *occurrence of an event* and *execution of an action* interchangeably since both refer to the same concept. When we refer to an *execution trace* or a *trace*, we mean a finite sequence of events according to Def. 1.

Although counterexamples are typically lengthy sequences of events, only a small number of events inside them are relevant to a property violation. In a concurrent system, the order of the events inside a counterexample can also be causal for the occurrence of a failure and can hence point to a bug. As we argue above, system failures are often due to an unexpected order of the occurrence of events in concurrent systems.

In this paper we explain concurrent counterexamples by identifying *explanatory* or *anomalous* sequences inside the counterexamples. Such sequences reveal specific orders between some events inside a counterexample which are presumed to be causal for the property violation.

Definition 2. *$\psi = \langle e_0, e_1, e_2, ..., e_m \rangle$ is a* subsequence *of $\sigma = \langle E_0, E_1, E_2, ..., E_n \rangle$ denoted as $\psi \sqsubseteq \sigma$, if and only if there exist integers $0 \leq i_0 < i_1 < i_2 < i_3 ... < i_m \leq n$ such that $e_0 = E_{i_0}, e_1 = E_{i_1}, ..., e_m = E_{i_m}$. We also call σ a super-sequence of ψ.*

Notice that a subsequence is not necessarily contiguous in the super-sequence. To capture the notion of a contiguous subsequence we introduce the concept of a *substring.*

Definition 3. *ψ is a* substring *of σ, if and only if there exist consecutive integers from $0 \leq i_0$ to $(i_0 + m) \leq n$ such that $e_0 = E_{i_0}, e_1 = E_{i_0+1}, ..., e_m = E_{i_0+m}$.*

Definition 4. *The sequence $\psi = \langle e_0, e_1, e_2, ..., e_m \rangle$ is an* explanatory sequence*, if for all execution traces σ, it holds that $\psi \sqsubseteq \sigma \Rightarrow \sigma \not\models \varphi$.*

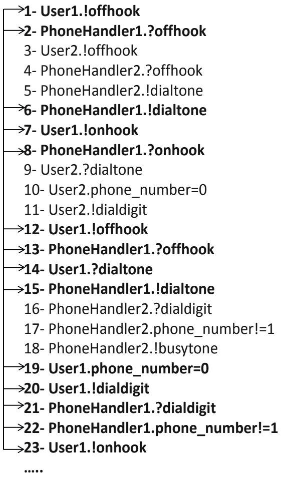
In the following subsection we will use a motivating example to illustrate that in concurrent systems such explanatory sequences occur in general as the *subsequences* of counterexamples. In our previous work, the sequences isolated for explaining counterexamples are the *substrings* of counterexamples containing contiguous events inside the counterexamples.

2.2 A Motivating Example

Using an example case study we now illustrate how a deadlock can occur due to the temporal order of execution of a set of actions in the model of a concurrent system. Referring to this example we then argue that contrasting sequential patterns of the bad and good datasets can reveal the anomalous sequences of actions that can help to explain the violation of a property, such as a deadlock in a concurrent system. We use the model of a preliminary design of a plain old telephony system (POTS)[1] as an example. This model was generated with the visual modeling tool VIP [10] and contains a number of deadlock problems. It comprises four concurrently executing processes corresponding to two users and two phone handlers. Each user in this model talks to a phone handler for making calls. The phone handlers are communicating with each other in order to switch and route user calls.

[1] The Promela code of the POTS case study is available at `http://www.inf.uni-konstanz.de/soft/tools/CEMiner/POTS7-mod-07-dldetect-never.prm`

A portion of a counterexample indicating the occurrence of a deadlock in the POTS model is given in Fig. 1. The events in this figure are displayed along with the name of the proctypes[2] to which they belong. The events are, in fact, Promela statements [9] that are separated by a "." from the name of the proctypes to which they belong. The events highlighted by the arrows on the left hand side of the trace reveal a problematic sequence of actions which can be interpreted as giving an explanation for the occurrence of a deadlock. This identified sequence for explaining the deadlock is, in fact, an example of an unforeseen interleaving of concurrent events. The presumed assumption of the model designer is that the *User1* and *PhoneHandler1* proctypes are synchronized so that when the *PhoneHandler1* proctype sends a *dialtone* message, the *User1* proctype subsequently receives it before taking any other action. However, as Fig. 1 indicates the model contains faults so that the events 6 and 15, which correspond to the sending of a "dialtone" message by the *PhoneHandler1* proctype, are not followed by a receiving event of the *User1* proctype. The statements executed by the User1 proctype after events 6 and 15 are $!onhook$ and $phone\_number = 0$, respectively, which causes an unread message to remain in the channel between the *User1* and *PhoneHandler1* proctypes. While the unread message of the event 6 is received by event 14, there is no corresponding receive event for the message of the event 15. Since the channels have a capacity of one message, the unread message of the event 15 causes the *PhoneHandler1* proctype to block after event 22 when it tries to send a "busytone" message to the *User1* proctype. Because of the blocking of the *PhoneHandler1* proctype, the *User1* proctype also blocks after the event 23. Due to the symmetry in the model, a similar interaction can occur between the *User2* and *PhoneHandler2* proctypes, which finally leads the system to a deadlock state.



1- User1.!offhook
2- PhoneHandler1.?offhook
3- User2.!offhook
4- PhoneHandler2.?offhook
5- PhoneHandler2.!dialtone
6- PhoneHandler1.!dialtone
7- User1.!onhook
8- PhoneHandler1.?onhook
9- User2.?dialtone
10- User2.phone_number=0
11- User2.!dialdigit
12- User1.!offhook
13- PhoneHandler1.?offhook
14- User1.?dialtone
15- PhoneHandler1.!dialtone
16- PhoneHandler2.?dialdigit
17- PhoneHandler2.phone_number!=1
18- PhoneHandler2.!busytone
19- User1.phone_number=0
20- User1.!dialdigit
21- PhoneHandler1.?dialdigit
22- PhoneHandler1.phone_number!=1
23- User1.!onhook
.....

Fig. 1. Part of a counterexample in POTS model

[2] *proctype* is the keyword used in Promela for defining a process.

One interesting characteristic of the explanatory sequence in Fig. 1 is that the events belonging to this sequence do not occur adjacently inside the counterexample. Instead they are interspersed with unrelated events belonging to the interaction of the *User2* and *Phonehandler2* proctypes. In general, events belonging to an explanatory sequence can occur at an arbitrary distance from each other due to the non-deterministic scheduling of concurrent events implemented in SPIN. From this observation, it can be inferred that the explanatory sequences are, in fact, subsequences of the counterexamples. In conclusion, we maintain that sequential pattern mining algorithms, which extract the frequent subsequences from a dataset of sequences without limitations on the relative distance of events belonging to the subsequences, are an adequate and obvious choice to extract explanatory sequences from large sets of counterexamples.

3 Counterexample Explanation

3.1 Generation of the Good and the Bad Datasets

In order to use sequential pattern mining and perform the contrast mining for explaining counterexamples we use the SPIN model checker to generate two sets of counterexamples, namely the "good" and the "bad" datasets. With the aid of the option "*-c0 -e*", which instructs SPIN to continue the state space search even when a counterexample has been found, we generate a set of counterexamples violating a given property φ, called the *bad dataset*, denoted by Σ_B: $\Sigma_B = \{\sigma \mid \sigma \not\models \varphi\}$. The *good* dataset includes the traces that satisfy φ. Such traces can be generated by producing counterexamples to $\neg\varphi$. This is justified by the following lemma:

Lemma 1. *For an execution σ, if σ satisfies φ, which is denoted as $\sigma \models \varphi$, then it holds that $\sigma \models \varphi \Leftrightarrow \sigma \not\models \neg\varphi$ [2].*

If φ is a safety property, the negation of this property yields a liveness property. The counterexamples violating a liveness property are infinite lasso shaped traces.

Definition 5. *Let $\hat{\phi}$ and (ϕ') denote finite traces. We call $\phi = \hat{\phi}.(\phi')^{\omega}$ an infinite lasso shaped trace where $\hat{\phi}$ is the finite prefix of ϕ and ω denotes that ϕ' is repeated infinitely.*

For the purpose of our analysis we produce finite traces from the infinite good traces by concatenating $\hat{\phi}$ with one occurrence of ϕ'. We use Σ_G to denote a good dataset: $\Sigma_G = \{\phi \mid \phi \models \varphi \wedge \phi \text{ is finite}\}$

3.2 Sequential Pattern Mining

We now give a brief overview of terminology used in sequential pattern mining, for a more detailed treatment we refer the interested reader to the cited literature and in particular to [4].

A sequence dataset S, $\{s_1, s_2, ..., s_n\}$, is a set of sequences. The *support* of a sequence α is the number of the sequences in S that α is a subsequence of: $support_S(\alpha) = |\{s \mid s \in S \wedge \alpha \sqsubseteq s\}|$. Given a minimum support threshold, *min_sup*, the sequence α is considered a sequential pattern or a frequent subsequence if its support is no less than *min_sup*: $support_S(\alpha) \geq min\_sup$. We denote the set of all sequential patterns mined from S with the given support threshold *min_sup* by FS_{S,min_sup}, i.e., $\mathrm{FS}_{S,min\_sup} = \{\alpha \mid support_S(\alpha) \geq min\_sup\}$.

Since mining all sequential patterns will typically result in a combinatorial number of patterns, some algorithms, such as [26,25] only mine *closed* sequential patterns. When a sequential pattern does not have any super sequence with the same support, it is considered as a closed pattern. The set of all closed sequential patterns mined from S with the given support threshold *min_sup*, denoted by CS_{S,min_sup}, is defined as follows:

Definition 6. *$CS_{S,min\_sup} = \{\alpha \mid \alpha \in FS_{S,min\_sup} \ \wedge \ \nexists\beta \in FS_{S,min\_sup}$ such that $\alpha \sqsubset \beta \wedge support_S(\alpha) = support_S(\beta)\}$.*

In fact, the support of a closed sequential pattern is different from that of its super-sequences. Since every frequent pattern is represented by a closed pattern, mining closed patterns leads to a more compact yet complete result set. In other words, closed patterns are the lossless compression of all the sequential patterns.

As an example, consider a sequence dataset S that has five sequences, $S = \{abced, abecf, agbch, abijc, aklc\}$. If the *min_sup* is specified as 4, $\mathrm{FS}_{S,4} = \{a : 5, b : 4, c : 5, ab : 4, ac : 5, bc : 4, abc : 4\}$ where the numbers denote the respective supports of the patterns. However, $\mathrm{CS}_{S,4}$ contains only two patterns, $\{abc : 4, ac : 5\}$.

For explaining counterexamples, we first mine closed sequential patterns from the bad and the good datasets with the given support thresholds T_B and T_G, respectively. We call the sets of closed patterns mined from the bad and the good datasets, $\mathrm{CS}_{\Sigma_B,T_B}$ and $\mathrm{CS}_{\Sigma_G,T_G}$, respectively. Contrasting the sequential patterns of the good and the bad datasets results in the patterns which are only frequent in the bad dataset. We call these patterns *anomalies*.

Definition 7. *We call $AS_{T_B,T_G} = \{\alpha \mid \alpha \in CS_{\Sigma_B,T_B} \wedge \alpha \notin CS_{\Sigma_G,T_G}\} = CS_{\Sigma_B,T_B} - CS_{\Sigma_G,T_G}$ the set of all anomalies.*

The anomalies computed according to Def. 7 are, in fact, a set of ordered sequences of events which give an explanation for the property violation. We maintain that the extracted set of anomalies is indicative of one or several faults inside the model. These anomalies can hence be used as the clues to the exact location of the faults inside the model and thereby greatly facilitate the manual fault localization process.

3.3 Complexity Issues

One of the major challenges in applying sequential pattern mining algorithms for explaining counterexamples is the scalability of these algorithms. In our precursory work [12] we discuss that mining sequential patterns from the datasets of counterexamples generated from typical concurrent system models is intractable. As we argue, this observation is due to inherent characteristics of those datasets, in particular the average length of the sequences that they include as well as their denseness. We conclude that we need some technique for reducing the length of the counterexamples in order to make the use of sequential pattern mining in this application domain tractable. We will propose a length reduction technique in the subsequent subsection.

Reducing the Length of the Traces. We are mainly analyzing the behavior of non-terminating communication protocols. By inspecting the structure of the finite traces of these protocols in Σ_B and Σ_G it becomes obvious that events belonging to particular processes, for instance some event a, may occur repeatedly. For example, inside a trace of the POTS model in Fig. 3 we can observe multiple executions of the actions *User1.!offhook* and *User1.!onhook*. In order to reduce the length of the execution traces in the good and the bad datasets, we exploit repetitions of the execution of actions inside the traces. Instead of analyzing the temporal order between all the events of a trace, we decompose each trace into a number of *subtraces* and examine the temporal order of the events that they contain in isolation. A possible choice for decomposing a trace into *subtraces* is via breaking the traces at the execution of a repeating action a. Thus, the obtained subtraces contain the events occurring between each two subsequent executions of a. We define the notion of a subtrace as follows:

Definition 8. *Let ϕ denote a finite trace and action* a *executed* n *times inside ϕ. By breaking ϕ at the executions of a, n* subtraces *will be generated. The $(i+1)^{th}$* subtrace *is defined as $\phi_{i+1,a} = \langle a_i, b_{i,0}, ..., b_{i,m}\rangle$ where a_i is the i^{th} execution of the action* a *in ϕ and $b_{i,j}$ is j^{th} event between the occurrence of a_i and a_{i+1}. The event that occurs next to $b_{i,m}$ is a_{i+1}.*

The subtraces $\phi_{i,a}$ reveal the temporal order between the events that are preceded by the execution of a in the traces, and hence by analyzing these subtraces we can only extract the anomalous sequences of events that precede the execution of a to explain counterexamples. Hence, the extracted anomalous sequences for explaining counterexamples will only contain one execution of the action a. Notice that as a consequence of this abstraction we lose access to the causes of failures that spread over multiple cycles, for instance the repeated occurrence of event a itself without the occurrence of some other event in between.

Instead of mining patterns from the datasets Σ_B and Σ_G, we mine patterns from the datasets Σ_{BR_a} and Σ_{GR_a} containing the subtraces of the traces in Σ_B and Σ_G, respectively: $\Sigma_{BR_a} = \{\sigma_a \mid \sigma_a \text{ is a subtrace of } \sigma \text{ and } \sigma \in \Sigma_B\}$ and $\Sigma_{GR_a} = \{\phi_a \mid \phi_a \text{ is a subtrace of } \phi \text{ and } \phi \in \Sigma_G\}$. In fact, for producing Σ_{BR_a}, we break up each trace in Σ_B and accumulate the resulting subtraces in Σ_{BR_a}. We do the same for Σ_{GR_a}. In analogy with Def. 7, anomalies are then computed by

$$\mathrm{AS}_{T_B,T_G,a} = \mathrm{CS}_{\Sigma_{BR_a},T_B} - \mathrm{CS}_{\Sigma_{GR_a},T_G}. \tag{1}$$

For instance, the identified sequence in the example of Sect. 2.2 for explaining the deadlock in the POTS model has portions $\langle 1, 2, 6, 7, 8\rangle$ and $\langle 12,\ 13,\ 14,\ 15,$

19, 20, 21, 22, 23⟩ which can be mined from the subtraces achieved by breaking the traces at the execution of *User1.!offhook*. As we have seen in Sect. 2.2, each of these portions reveals a problematic sequence of actions that gives clues about the location of the fault in the model.

As we will see in the experimental results section, this reduction technique can reduce the average sequence length of the datasets significantly, and hence can make mining sequential patterns from them feasible. Table 1 shows the amount of the length reduction for the bad and good datasets of the POTS model obtained by applying this length reduction technique.

Table 1. Average sequence length before and after reduction, POTS model datasets

Model	Datasets	#seq. before reduction	#seq. after reduction	avg. seq. len. before reduction	avg. seq. len. after reduction
POTS	bad	4109	497595	1677	13
	good	107029	43668	3079	21

Determining an action a at which to break up the traces is a heuristic decision. In principle, any action whose execution is recurrent inside the execution traces can be used for breaking up the traces. However, considering the functionality of the model some actions may seem to be more interesting to be analyzed with respect to their ordering relationships with other actions. Such actions of interest can correspond, for instance, to the start of interactions between different concurrent processes in a communication protocol. For example, in the POTS model many interactions start with the execution of *User1.!offhook*. It initiates a sequence of events handling a telephone call and is hence a candidate for the event a. Apparently, we lose some ordering relationships between the actions of a model by shortening the traces via breaking them at the execution of some specific action. However, if we use the actions corresponding to the start of interactions between concurrent processes for breaking the traces, we may lose less important temporal orders from the user perspective. Currently, in our case studies we detect the first action that is taken by one of the processes in the system and use it for breaking up the traces. An alternative strategy for determining the action to break up the traces is by calculating how much reduction can be gained on the average from each individual action, and then to choose the one with the highest reduction ratio. Another heuristic is choosing those actions which divide the traces evenly or result in subtraces with similar length. For example, Table 2 shows different amount of length reduction gained from different actions in the POTS model. In the experimental results section, we report on the results achieved by breaking the traces at actions *U1.!offhook (117)* and *P1.?offhook* which give us the most length reduction.

Threats to Validity. It should be noted that this reduction technique is mainly applicable to execution traces that include repeating patterns of execution of actions, such as non-terminating communication protocols. For some large models

Table 2. Length reduction for different actions in the bad dataset of POTS model, U1 and P1 refer to User1 and PhoneHandler1 processes, respectively.

Action line no.	U1.!offhook 117	U1.?ringtone 101	U1.!offhook 146	U1.!onhook 351	U1.!onhook 294	P1.?offhook 464
avg. seq. len.	14	343	1442	548	47	13

the proposed reduction technique may still not sufficiently reduce the length of the execution traces. As we have seen the produced anomalies for explaining counterexamples only contain one execution of the action a. If however for understanding the cause of the property violation inside the counterexample, the isolation of an ordered sequence of events containing more than one execution of a is required, then the analysis of the subtraces would not be sufficient. In other words, since we lose some temporal order by analyzing only the subtraces, we may not be able to explain some concurrency bugs.

3.4 Contrasting Sequential Patterns

For mining closed sequential patterns we use an algorithm called CloSpan [26]. The flowchart of our method is given in Fig. 2.

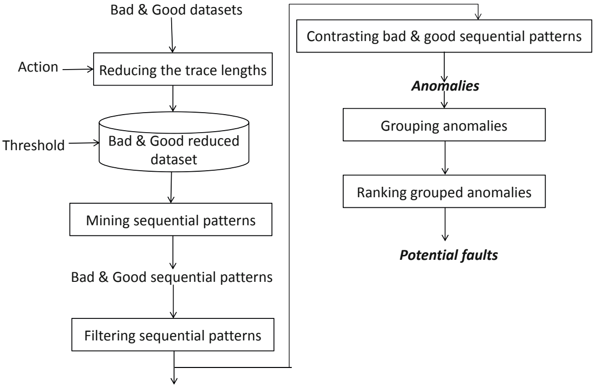
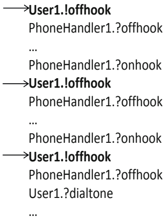

Bad & Good datasets
Action
Reducing the trace lengths
Threshold
Bad & Good reduced dataset
Mining sequential patterns
Bad & Good sequential patterns
Filtering sequential patterns
Contrasting bad & good sequential patterns
Anomalies
Grouping anomalies
Ranking grouped anomalies
Potential faults

Fig. 2. Flowchart of explaining counterexamples method

The final result set of the method contains the distinguishing patterns representing the set of sequences of actions that are only frequent or typical in the bad dataset. This set is generated by equation (1). The user defined threshold values, T_B and T_G in equation (1) are, in fact, the parameters of our method. By decreasing the value of the support threshold, the number of the generated sequential patterns from a dataset of traces increases. In order to reduce the number of the mined patterns, we remove the patterns which are *substrings* of some other generated pattern. This is because the ordering relationship that can be inferred from these patterns can also be inferred from the longer patterns that these patterns are substrings of.

In order to facilitate the interpretation of the result set obtained by equation (1) we divide the anomalies into a number of groups so that each group contains patterns which are all subsequences of the longest pattern in that group. Fig. 4 shows an example of such a group of patterns. One temporal order that can be inferred from the longest pattern in Fig. 4 is $\langle 334, 1426, 444\rangle$. From the subsequences of the longest pattern, it can be inferred that not always "1426" occurs between "334" and "444" because $\langle 334, 444\rangle$ is also frequent, and not always "1426" is preceded by "334" because $\langle 1406, 1426, 444\rangle$ is also frequent.



```
→User1.!offhook
 PhoneHandler1.?offhook
 ...
 PhoneHandler1.?onhook
→User1.!offhook
 PhoneHandler1.?offhook
 ...
 PhoneHandler1.?onhook
→User1.!offhook
 PhoneHandler1.?offhook
 User1.?dialtone
 ...
```

Fig. 3. Multiple occurrence of events inside an execution trace

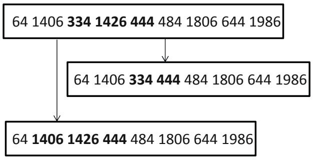


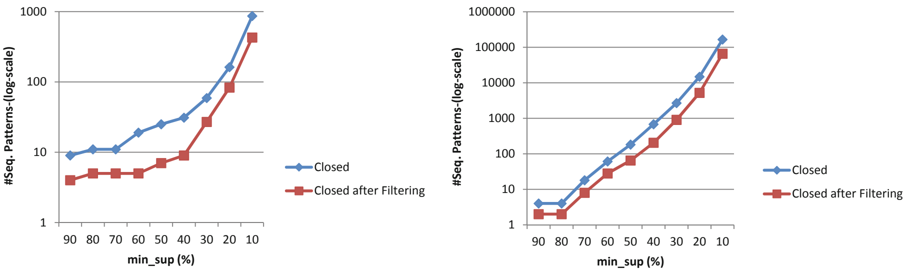
Fig. 4. Patterns inside one group

The groups of patterns are then ordered based on the length of the longest pattern inside them. Groups with the shorter length of the longest pattern will be ranked higher because the analysis of these patterns by the user requires less effort.

4 Experimental Evaluation

The experiments that we report on in this section were performed on a 2.67 GHz PC with 8 GB RAM and Windows 7 64-bit operating system. The prototype implementation of our method was realized using the programming language C#.Net 2010. We discuss the results obtained by applying our method to a number of case studies.

Case Study 1: POTS Model. We first applied our method to the POTS model (see Sect. 2.2) in order to obtain explanations for the occurrence of deadlocks. The execution traces were shortened in length by breaking the original traces at the execution of the action *User1.!offhook* as it has been explained in Sect. 3.3. In order to study the effect of the threshold value on the number of the generated patterns in the result set we applied different threshold values, starting with a comparatively high threshold value of 90%. Fig. 5 shows how the number of the generated patterns is reduced after our filtering step. The reduction is by a factor of approximately 0.5. It also illustrates how the number of the closed sequential patterns increases when decreasing the threshold. Mining closed sequential patterns from the good dataset of POTS with the min_sup of 10% takes 359.651 sec. and consumes 31.327 MB of main memory while with the min_sup of 90% it takes only 0.074 sec. and consumes only 3.69 MB of main memory.



(a) Bad dataset

(b) Good dataset

Fig. 5. Number of the closed sequential patterns in the bad and good datasets before and after filtering

In Fig. 6, the number of the anomalies obtained by equation (1) along with the number of the groups that these anomalies are divided into are given. From the figures 5 and 6, it can be inferred that although the number of the generated closed sequential patterns from the bad and good datasets can be quite high, the number of the anomalies that the user needs to inspect to understand the root cause of the deadlock is mostly less than 10, at least for thresholds of not less than 20. In Fig. 6, the precision of the method shows the number of the sequences in the result set which actually reveal some anomalous behavior. As this figure shows, only for the thresholds of 30%, 20% and 10% the precision is less than 100%. Considering the way that we generate the good and the bad datasets, these datasets may not include all the possible good and bad traces that can be produced by the execution of the model. In the final result set of the method, therefore, we may get some false positives that do not reveal any problematic behaviors in the model. The computed precision measure for each case study shows the number of the true anomalous sequences among all the sequences of the result set. This precision was calculated manually.

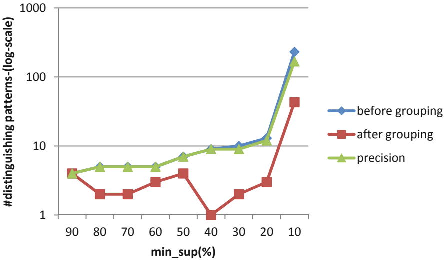


Fig. 6. Number of the anomalies, number of the groups of anomalies and the precision

The manual inspection of the anomalous sequences in the result set of the method reveals some faults in the model. In fact, two faults can be detected from the result sets generated by the thresholds 20% and 10%. Other result sets which are generated by higher threshold values only reveal one fault. For example, one of the anomalous sequences for the support threshold of 90% is *User1.!offhook, PhoneHandler1.?offhook, User1.?dialtone*, while according to the behavior of the model the expected sequence from the user perspective is *User1.!offhook, PhoneHandler1.?offhook, PhoneHandler1.!dialtone, User1.?dialtone.* Considering the expected sequence a receiving *dialtone* message should always be preceded by a sending *dialtone* message. The anomalous sequence reveals a deviation from the expected sequence because in this sequence the receiving *dialtone* messages is not preceded by a corresponding sending *dialtone* message. This implicitly reveals the presence of an unread message in the channel. Finally, it can be inferred that there is a lack of synchronization between the user and the phone handler proctypes so that when the phone handler sends a *dialtone* message, the user instead of receiving that message takes another action.

By breaking the traces at *PhoneHandler1.?offhook* instead of *User1.!offhook*, by the support threshold of 90% a result set containing of 5 anomalies will be generated. In fact, these anomalies also reveal the same fault as explained above.

It must be noted that our method is not supposed to be complete, and we use the method as part of an iterative debugging process. After each run of the method, aided by the revealed anomalous sequences the user will try to remove as many causes of property violation as possible. In case the model still contains faults after being modified, the user will apply the method again. This procedure can be iterated until all the causes of property violation in the model have been removed. For example, we tried to remedy the problem in POTS by adding some code in the user proctype which removes a message *dialtone* from the channel between the user and the phone handler proctypes, if it is present, when sending an *onhook* message. After this modification, we again applied our method on the resulting model, this time the number of the generated counterexamples decreased from 4109 to 2229. The produced result set reveals that there is still a lack of synchronization between the user and the phone handler proctypes.

Case Study 2: Rether Model. The second model is a Real-time Ethernet protocol named Rether. It was obtained from [21]. In order to reduce the size and complexity of the original model from [21] we have reduced the values of its parameters. A detailed description of this model can be found in [12]. We applied our method to this model in order to explain the occurrence of a deadlock. The statement "i=0" of the Token proctype was used for breaking the execution traces because the interaction between the processes in this model starts with the execution of this statement. Table 3 shows the extent of the length reduction of the traces for this case study.

Table 3. Results of length reduction in the Rether model datasets

datasets	#seq. before reduction	#seq. after reduction	avg. seq. len. before reduction	avg. seq. len. after reduction
bad	8	92	322	28
good	78	812	298	29

A threshold value of 2% was applied to the reduced length bad and good datasets for generating the sequential patterns. In Table 4 the number of the generated sequential patterns before and after filtering along with the number of the anomalies obtained by equation (1) and the number of the faults detected by the user inspection are given.

Table 4. Rether model results

datasets	#seq. patterns	#seq. patterns after filtering	#anomalies	#groups of anomalies	precision	#detected faults
bad	182	170	23	11	7	1
good	466	244				

Even though appr. 65% of the extracted groups reveal some problematic behavior in the system, the inspection of only 2 of them, corresponding to the first and the 8^{th} groups in the ranked result set, is required for localizing an atomicity violation in one of the proctypes of the model. Due to space limitations we refer the interested reader to our previous work [12] for an extensive discussion of which specific sequence of actions reveals an atomicity violation in this model.

Comparison with Our Previous Work. The fault localization method that we proposed in a precursory paper [12] aids the user in locating unforeseen interleavings inside the counterexamples of concurrent systems by extracting a set of short substrings of mainly length two that only occur in the bad dataset. These short substrings along with the corresponding counterexamples are given to the user for further analysis. For example, for this case study, this method generates 3 short distinguishing substrings of length two which are given to the user along with the corresponding counterexamples. With the aid of these substrings, the user needs to inspect on the average 30 events inside the corresponding counterexamples in order to identify the anomalous sequences pointing to an atomicity violation bug in the model. However, the anomalous sequences detected with the aid of the method proposed in this paper are in themselves indicative of the atomicity violation bug in the model. In other words, as opposed to our precursory work an inspection of counterexamples is not required at all. Specifically, in order to detect an atomicity violation in this case study, an anomalous sequence of at least length 30 needs to be isolated inside a counterexample. With the aid of the short substrings of length 2 extracted by our previous method, the user still needs to inspect the counterexample in order to isolate an anomalous sequence of length 30, even though these substrings facilitate the user inspection greatly. However, the groups of anomalies generated by the method of this paper contain the anomalous sequence of length 30 required for locating the atomicity violation in the model. In fact, the last 7 events of this sequence appear in the first group of the ranked result set and the rest of the events are included in the 8^{th} group. We contend that the current method imposes less inspection effort on the user for locating the faults in the model.

Table 5. Railway model results

datasets	#seq.	avg. seq. len.	#seq. patterns	#seq. patterns after filtering	#anomalies	#detected faults
bad	28	15	1	1	1	1
good	85	15	6	2		

Case Study 3: Railway Model. We finally applied our method to explain counterexamples indicating the violation of a safety property in the small railroad crossing example which is also used as a sample case study in [11]. The desired safety property is that the car and the train should never be in the crossing simultaneously, which is considered a hazardous state of the system. In this small model, the length reduction step was not necessary.

Table 5 summarizes all the figures related to this model and the achieved results by applying the high support threshold value of 90%. The detected anomalous sequence reveals a sequence of actions that leads the system to an undesired state in which the variables *"carcrossing"* and *"traincrossing"* have both the value "1". This indicates that both a car and a train are in the crossing at the same time, which is equivalent to a hazard state. This sequence, in fact, guides the user to the location of an atomicity violation bug in the *"Gate"* proctype. The presumed intention of the model designer is that the transmission of the signal "1" through the *gateCtrl* channel would be performed atomically with the changing of the global variable *"gatestatus"* to "1". However, due to the fault in the model, the execution of these two statements is interleaved with some other concurrent actions and leads the system to a hazard state.

Dataset Generation. As it has been explained in Sect. 3, we use the option "*-c0 -e*" in SPIN, for generating the good and the bad datasets which can be time-consuming for some case studies. For example, for the POTS model SPIN generates 303,589 good traces which takes around 14 hours. However, the dataset generation for the other two case studies takes less than a minute. If the generated datasets have fewer numbers of traces than the ones generated with the option "*-c0 -e*", our method is still applicable to them since the method is not guaranteed to be complete. However, when the datasets offer a higher coverage of the good and the bad behaviors, the output of the method is more precise and the number of the false positives among the explanations is reduced.

5 Related Work

In this section, we briefly discuss closely related work that has not yet been addressed in earlier sections.

Pattern Mining in Software Analysis. Data mining techniques have proven to be useful in the analysis of very large amounts of data produced in the course of different activities during various states of the software system development cycle. Frequent pattern mining techniques which find commonly occurring patterns in a dataset are broadly used for mining specifications and localizing faults in program code [15,14,19,5,22]. The work documented in [15] adapts sequential pattern mining techniques in order to mine specifications from recorded traces of software system executions. It seems that the patterns generated by this method can also be used for counterexample explanation. However, we faced scalability issues when applying this method to the POTS model case study that we introduce in Section 2.2. The longest distinguishing patterns between the bad and the good datasets that could be generated by this method were only 2 events long and did not carry any interesting information with respect to ordering relationships amongst events. CHRONICLER [22] is a static analysis tool which infers function precedence protocols defining ordering relationships among function calls in program code. For extracting these protocols a sequence mining algorithm is used. The methods in [14,19,5] use graph and tree mining algorithms for localizing faults in sequential program code. A commonality of these methods is that they first construct behavior graphs such as function call graphs from execution traces. They then apply a frequent graph or tree mining algorithm on the passing and failing datasets of constructed graphs in order to determine the suspicious portions of the sequential program code. As opposed to this approach, our goal is to identify sequences of interleaved actions in concurrent systems, which the above cited works are unable to provide.

Concurrency Bug Detection Methods. AVIO [16] only detects atomicity violations and, as opposed to our method, is tailored to only identify single variable bugs. Examples of tools which only focus on detecting data races are lockset bug detection tools [23] and happens-before bug detection tools [18]. In contrast to these approaches, which lack generality and rely on heuristics that are specific to a class of bugs, the output of our method in the form of anomalous sequences can be indicative to any type of concurrency bugs in the program design that can be characterized by a reachability property.

The work described in [17] proposes a more general approach for finding concurrency bugs based on constructing context-aware communication graphs from execution traces. Context-aware communication graphs use communication context to encode access ordering information. A key challenge of this method is, however, that if the relevant ordering information is not encoded, bugs may not lead to graph anomalies and therefore remain undetected. Our method does not rely on such an encoding but directly analyzes the temporal ordering of the event. It therefore appears to be more general than the approach in [17].

Counterexample Explanation Methods. In [12], we provide a detailed comparison of our method with a closely related work by Groce and Visser [7]. For that comparison, the arguments given in [12] are also valid for our current work, because, in fact, the current method is the enhancement of our precursory work. The *causality checking* method proposed in [11] computes automatically the causalities in system models by adapting the counterfactual reasoning based on the *structural equation model (SEM)* by Halpern and Perl [8]. This method identifies sequences of events that cause a system to reach a certain undesired state by extending depth-first search and breadth-first search algorithms used for a complete state space exploration in explicit-state model checking. It seems that the main superiority of our method is less computational cost in terms of memory and running time for detecting at least one fault in the model. The *causality checking* method considers all the possible finite good and bad execution traces for identifying the combination of events which are causal for the violation of a safety property. Since we do not seek completeness, our mining method is still applicable even if the datasets do not include all the possible good and bad execution traces, which can be an impediment in practice.

Some other automated counterexample explanation techniques such as [3,24,6] only take the values of program or model variables into account when computing which variable values along a counterexample trace cause a violation of some desired property. In contrast, the method we propose here considers the order of execution of actions and can hence explain property violations which are due to a specific order of execution of actions.

6 Conclusion

We have presented an automated method for the explanation of model checking counterexamples for concurrent system models. From a dataset of counterexamples we extract a number of anomalous sequences of actions that prove to point to the location of the fault in the model by leveraging a frequent pattern mining technique called sequential pattern mining. An experimental analysis showed the effectiveness of our method for a number of indicative deadlock checking case studies.

In future work we plan to reduce the computational effort that our method entails by imposing a limit on the number of context switches in generation of the good and the bad traces.

Acknowledgements. We wish to gratefully acknowledge a careful review of this work by Georg Weissenbacher.

References

1. Agrawal, R., Srikant, R.: Mining sequential patterns. In: ICDE (1995)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Massachusetts (2008)
3. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009)
4. Dong, G., Pei, J.: Sequence Data Mining. Springer (2007)
5. Fatta, G.D., Leue, S., Stegantova, E.: Discriminative pattern mining in software fault detection. In: Proceedings of the 3rd International Workshop on Software Quality Assurance (2006)
6. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. In: International Journal on Software Tools for Technology Transfer (STTT) (2006)
7. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–135. Springer, Heidelberg (2003)

8. Halpern, J., Pearl, J.: Causes and explanations: A structural-model approach. part I: Causes. The British Journal for the Philosophy of Science (2005)
9. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addision-Wesley (2003)
10. Kamel, M., Leue, S.: VIP: A visual editor and compiler for v-promela. In: Graf, S., Schwartzbach, M. (eds.) TACAS/ETAPS 2000. LNCS, vol. 1785, pp. 471–486. Springer, Heidelberg (2000)
11. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 248–267. Springer, Heidelberg (2013)
12. Leue, S., Tabaei Befrouei, M.: Counterexample explanation by anomaly detection. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 24–42. Springer, Heidelberg (2012)
13. Lewis, D.: Counterfactuals. Wiley-Blackwell (2001)
14. Liu, C., Yan, X., Yu, H., Han, J., Yu, P.S.: Mining behavior graphs for backtrace of noncrashing bugs. In: Proceedings of the Fifth SIAM International Conference on Data Mining (2005)
15. Lo, D., Khoo, S., Liu, C.: Efficient mining of iterative patterns for software specification discovery. In: KDD (2007)
16. Lu, S., Tucek, J., Qin, F., Zhou, Y.: Avio: Detecting atomicity violations via access interleaving invariants. In: ASPLOS (2006)
17. Lucia, B., Ceze, L.: Finding concurrency bugs with context-aware communication graphs. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (2009)
18. Netzer, R., Miller, B.: Improving the accuracy of data race detection. In: Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming. ACM Press (1991)
19. Parsa, S., Naree, S.A., Koopaei, N.E.: Software fault localization via mining execution graphs. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part II. LNCS, vol. 6783, pp. 610–623. Springer, Heidelberg (2011)
20. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: 17th International Conference on Data Engineering (ICDE 2001) (2001)
21. Pelanek, R.: Benchmarks for explicit model checkers (2006), `http://anna.fi.muni.cz/models`
22. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: Proceedings of the 29th International Conference on Software Engineering (ICSE) (2007)
23. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems (TOCS) 15(4) (1997)
24. Wang, C., Yang, Z., Ivančić, F., Gupta, A.: Whodunit? causal analysis for counterexamples. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 82–95. Springer, Heidelberg (2006)
25. Wang, J., Han, J.: Bide: Efficient mining of frequent closed sequences. In: ICDE (2004)
26. Yan, X., Han, J., Afshar, R.: Clospan: Mining closed sequential patterns in large datasets. In: Proceedings of 2003 SIAM International Conference on Data Mining (SDM 2003) (2003)
27. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, Burlington (2009)