Counterexample Explanation by Anomaly Detection

Stefan Leue and Mitra Tabaei Befrouei

Department of Computer and Information Science
University of Konstanz
D-78457 Konstanz, Germany
Email: {Stefan.Leue|Mitra.Tabaei}@uni-konstanz.de

Abstract. Since counterexamples generated by model checking tools are only symptoms of faults in the model, a significant amount of manual work is required in order to locate the fault that is the root cause for the presence of counterexamples in the model. In this paper, we propose an automated method for explaining counterexamples that are symptoms of the occurrence of deadlocks in concurrent systems. Our method is based on an analysis of a set of counterexamples that can be generated by a model checking tool such as SPIN. By comparing the set of counterexamples with the set of correct traces that never deadlock a number of sequences of actions are extracted that aid the model designer in locating the cause of the occurrence of a deadlock. We first argue that the obvious approach to extract such sequences is by sequential pattern mining and by contrasting patterns that are typical for the deadlocking counterexample traces but not typical for non-deadlocking traces fails due to the inherent complexity of the problem. We then propose to extract substrings of specific length that only occur in the set of counterexamples for explaining the occurrence of deadlocks. We use a number of case studies to show the effectiveness of our approach and to compare it with an alternative approach to the counterexample explanation problem.

Keywords: model checking, deadlocks, counterexample explanation, anomaly detection, concurrency bugs

1 Introduction

Model checking is an established technique for the automated analysis of hardware and software systems. A model checker systematically checks whether a formal model M of the system satisfies a formalized property P [2]. If M contains a fault so that M does not satisfy P, as a symptom of the fault in the model the model checker generates a counterexample to the satisfaction of P. Given that counterexamples are only symptoms of faults in the model, a significant amount of manual analysis is required in order to locate a fault that constitute a root cause for the presence of the counterexample in the model. Model designers need to inspect lengthy counterexamples of sometimes up to thousands of events in order to understand the cause of the violation of P by M. Since this manual inspection is time consuming and error prone, an automatic method for explaining counterexamples that assist 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 occurrence of deadlocks in concurrent systems. Our method is based on an analysis of a set of counterexamples that can be generated by a model checking tool such as SPIN [14]. When SPIN explores exhaustively the state space of a model in order to locate all property violating states, it can generate a set of counterexamples. We refer to the set of counterexamples that show how the model violates a 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 property. We refer to this set of non-violating traces as the *good dataset*.

By examining the differences in the traces of the good and bad datasets, we extract a number of sequences of actions that aid the model designer in locating the cause of the occurrence of a deadlock. Since the extracted sequences of actions are those that are common in the bad dataset but not common in the good dataset, we refer to them as anomalies. In fact, examining the differences between faulty and successful runs is a widely used approach for locating faults in the program codes [26]. Lewis' theory of causality and counterfactual reasoning provides justification for these type of fault localization approaches [16].

A widely adopted paradigm for the semantics of concurrent systems is that of an *interleaving*, which gives rise to a nondeterministic choice between activities of the concurrently executing processes [2]. In fact, the interleaving semantics determines in which order the actions of the processes that run concurrently in the system are executed. System designers tend to think sequentially when designing the model of a system. In concurrent systems it is therefore highly probable that they have not foreseen some interleavings that their model encompasses. As a consequence, one of the main sources of failure in concurrent systems is unforeseen interleavings resulting in undesired behavior or unexpected results [2]. The anomalies produced by our method, which are given in the form of sequences of actions, can reveal to the model designer unforeseen interleavings that lead the system to a deadlock state. Deadlocks occur in a concurrent system when processes wait in a circular, non-preemptive fashion on each other and cannot make progress [13]. Proving the absence of deadlocks is one of the first sanity checks undertaken in the analysis of concurrent systems.

Related Work. There are a number of works on automatically explaining counterexamples using different technical approaches and with different objectives. The work documented in [5] using the notion of *causality* introduced by Halpern and Pearl [12] formally defines a set of causes for the failure of a property on a given counterexample trace. For the explanation of a counterexample, this method deals with what values on the counterexample cause it to falsify the property. In [22] Wang et al. focus on explaining the class of assertion violation failures. Their method uses an efficient weakest precondition algorithm which is executed on a single concrete counterexample in order to extract a minimal set of contradicting word-level predicates. Groce et al. [10] developed a tool called *explain*, which extends the CBMC model checker [15], for assisting users in understanding and isolating errors in ANSI C programs based on Lewis' counterfactual causality reasoning. Given a counterexample, *explain* finds the most similar successful execution based on a *distance metric* on execution traces. The differences (Δ s) between the most successful execution and the counterexample, after being refined by a slicing step, is given to the programmer as an explanation. The distance between executions a and b is measured based on the number of the variables to which a and b assign different values. In contrast to the three methods cited above, our counterexample analysis method does not consider any values that are assigned to variables, instead only the order of execution of actions inside execution traces are taken into account. Therefore, we are able to give explanation to counterexamples in which the violation of a property is due to a specific *order* of execution of actions. Moreover, the other methods are based on an analysis of one single counterexample while in our method for extracting commonalities we use non-singleton sets of counterexamples.

The work by Ball et al. [3] compares a counterexample with a set of similar correct traces in order to extract single program statements that are only executed in the counterexample. These program statements are reported to the user as the suspicious parts of the program code that are likely to be the cause of the violation of the property. In this method, if a counterexample violates a property at some control location c of the program code, then the execution traces that reach to c without violating the property are considered as similar correct traces. The method has been implemented in the context of the *SLAM* project in which a software model checker that automatically verifies temporal safety properties of C programs has been developed [4]. Since this method only considers single program statements, it cannot express counterexamples in which the violation of a property is due to a specific *order* of execution of actions. The criteria they use for finding similar correct traces are similar to those used by the method in [11]. In fact, the method in [11] is most closely related to ours, so we provide a detailed comparison of this method with ours in Sect. 6.

There are a few fault localization techniques based on testing which are analogous to ours and consider the actual order of execution of the statements in the program in order to locate the fault in the program code [18] [6]. The work of [6] had an important influence on our method.

Structure of the Paper. Section 2 presents a detailed example to show how an unforeseen interleaving can cause a deadlock to occur in the model of a concurrent system. Section 3 argues that a sequential pattern mining based approach for extracting sequences that can explain the occurrence of a deadlock will fail due to the inherent complexity of the problem. Section 4 describes our proposed method based on an extraction of substrings of a specific length that only occur in the set of counterexamples for explaining the occurrence of deadlocks. We then present the experimental results in Section 5, followed by a detailed comparison of our method with the work by Groce and Visser [11] in Section 6. Finally Section 7 concludes with a note on future work.

2 A Motivating Example

In this section, using an example case study we illustrate how a deadlock can occur due to the specific order of execution of a set of actions in the model of a concurrent system. The model we use in this example is taken from the BEnchmarks for Explicit Model checkers (BEEM) [20]. It is a Real-time Ethernet protocol named Rether. This protocol is a contention-free token bus protocol for the data-link layer of the ISO protocol stack. Its purpose is to provide guaranteed bandwidth, deterministic and periodic network access to multimedia applications over commodity Ethernet hardware. In order to make the original model taken from [20] smaller and simpler, we have reduced the values of its parameters as follows:

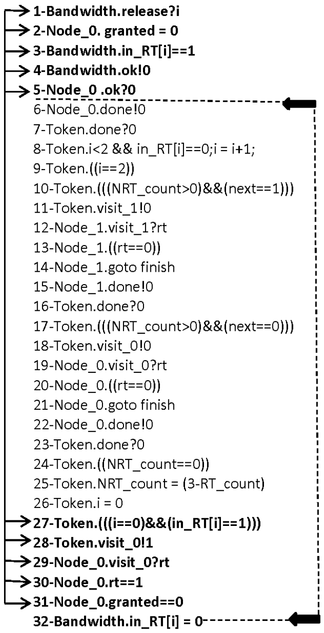
N = 2	Number of the nodes
Slots = 3	Number of slots (a bandwidth)
RT_slots = 1	Maximum number of slots for real-time transmission (should be smaller than Slots)

The Promela code of this model consists of three proctypes:

1. The *Bandwidth* proctype, which manages the access of the nodes to the real-time transmission. It allocates and frees the real-time transmission slots upon receiving *reserve* and *release* messages from the nodes.
2. The *Token* proctype, which guarantees deterministic and periodic access to the bandwidth by handing in a token to the nodes in turn.
3. The *Node* proctype, which corresponds to a node in the protocol. It communicates with the *Token* and *Bandwidth* proctypes in order to access the bandwidth slots. In our example, only two instances of this proctype, which are named *Node_0* and *Node_1*, are created at run time.

In Fig. 1, the last 32 events of a counterexample with 72 events which shows how the Rether model goes to a deadlock state are given. The events in this figure are displayed along with the name of the proctypes to which they belong. The events are, in fact, Promela statements [14]. The name of the events are separated by a "." from the name of the proctype to which they belong.

By manual inspection and using knowledge of the functioning of the model we can identify a subsequence of 10 events of the counterexample that can explain the occurrence of the deadlock. These 10 events are highlighted by arrows on the left hand side of the trace in Fig. 1. In order to understand how this subsequence leads the system into a deadlock state we need to inspect the parts of the Promela code of the model which include the statements corresponding to the 10 events identified above. These parts are given in Fig. 2 in which the statements corresponding to the spotted 10 events are displayed in bold font. The numbers inside parenthesis in front of these statements show the number of the corresponding event from Fig. 1. When events 4 and 5 in Fig. 1, which correspond to line 7 of the *Bandwidth* and line 20 of the *Node_0* proctypes, are executed, line 8 of the *Bandwidth* and line 17 of the *Node_0* proctypes become enabled simultaneously. In the trace from Fig. 1, line 17 of the *Node_0* proctype, which corresponds to event 6 in this figure is chosen for execution. Following the



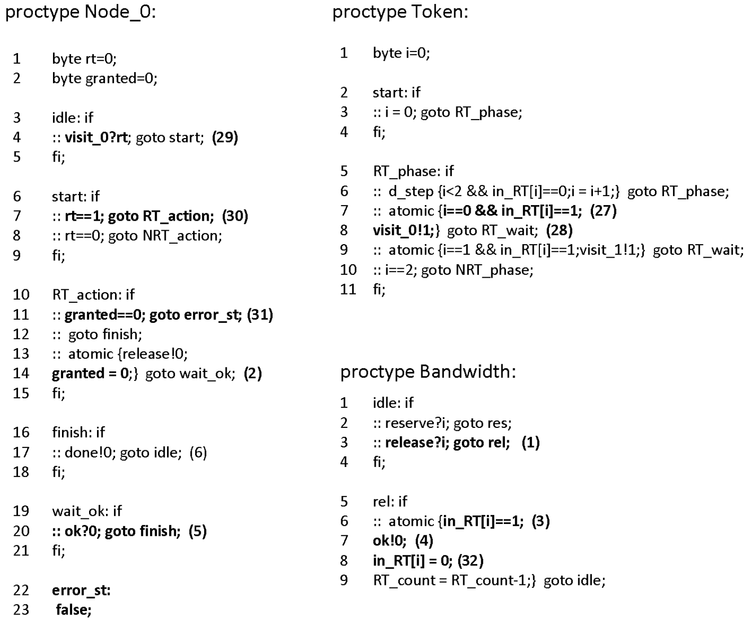
```
1-Bandwidth.release?i
2-Node_0. granted = 0
3-Bandwidth.in_RT[i]==1
4-Bandwidth.ok!0
5-Node_0 .ok?0
6-Node_0.done!0
7-Token.done?0
8-Token.i<2 && in_RT[i]==0;i = i+1;
9-Token.((i==2))
10-Token.(((NRT_count>0)&&(next==1)))
11-Token.visit_1!0
12-Node_1.visit_1?rt
13-Node_1.((rt==0))
14-Node_1.goto finish
15-Node_1.done!0
16-Token.done?0
17-Token.(((NRT_count>0)&&(next==0)))
18-Token.visit_0!0
19-Node_0.visit_0?rt
20-Node_0.((rt==0))
21-Node_0.goto finish
22-Node_0.done!0
23-Token.done?0
24-Token.((NRT_count==0))
25-Token.NRT_count = (3-RT_count)
26-Token.i = 0
27-Token.(((i==0)&&(in_RT[i]==1)))
28-Token.visit_0!1
29-Node_0.visit_0?rt
30-Node_0.rt==1
31-Node_0.granted==0
32-Bandwidth.in_RT[i] = 0
```

Fig. 1. The last 32 events of a counterexample in the Rether model

execution of event 30 in Fig. 1, corresponding to line 7 of the *Node_0* proctype, control is transfered to line 10 of this proctype which is an *if* statement. Lines 11 and 12 of this *if* statement are enabled simultaneously since line 12 is a *goto* statement and the guard of line 11, *granted == 0*, is true. The value of *granted* is set to zero at event 2 in Fig. 1 and remains unchanged up to event 30. As Fig. 1 shows, if line 11, which corresponds to event 31 in this figure, is executed, then a deadlock will occur.

One interesting characteristic of the identified subsequence in Fig. 1 is that the 10 events belonging to it do not occur adjacently inside the counterexample. While the first and the last five events occur next to each other, between these two groups of events there is a gap of 21 events. This is due to the non-deterministic scheduling of concurrent events due to the interleaving semantics implemented in SPIN. As we have seen above, although line 8 of the *Bandwidth* proctype was enabled after event 5, due to the non-deterministic execution of concurrent actions its execution is deferred to step 32. Dashed lines and thick arrows on the right hand side of the Fig. 1 illustrate the gap between the position in the trace in which the statement *Bandwidth.in_RT[i]=0* becomes enabled, and the position in which it is actually executed.

The identified subsequence in Fig. 1 explaining the deadlock is an example of an unforeseen interleaving. The presumed intention of the model designer is that event 5 and 32 be executed in an atomic step, which means they could



```
proctype Node_0:

byte rt=0;
byte granted=0;

idle: if
:: visit_0?rt; goto start;  (29)
fi;

start: if
:: rt==1; goto RT_action;  (30)
:: rt==0; goto NRT_action;
fi;

RT_action: if
:: granted==0; goto error_st; (31)
::  goto finish;
::  atomic {release!0;
granted = 0;}  goto wait_ok;  (2)
fi;

finish: if
:: done!0; goto idle;  (6)
fi;

wait_ok: if
:: ok?0; goto finish;  (5)
fi;

error_st:
 false;
```

```
proctype Token:

byte i=0;

start: if
:: i = 0; goto RT_phase;
fi;

RT_phase: if
::  d_step {i<2 && in_RT[i]==0;i = i+1;}  goto RT_phase;
::  atomic {i==0 && in_RT[i]==1;  (27)
visit_0!1;}  goto RT_wait;  (28)
::  atomic {i==1 && in_RT[i]==1;visit_1!1;}  goto RT_wait;
:: i==2; goto NRT_phase;
fi;
```

```
proctype Bandwidth:

idle: if
:: reserve?i; goto res;
:: release?i; goto rel;   (1)
fi;

rel: if
::  atomic {in_RT[i]==1;  (3)
ok!0;  (4)
in_RT[i] = 0; (32)
RT_count = RT_count-1;}  goto idle;
```

Fig. 2. Parts of the Promela code of the Rether model

not be interleaved with the actions of other proctypes. However, the proctype was implemented in a faulty way, so that its concurrent execution with other proctypes allowed the two mentioned events to be executed as a non-atomic sequence of events, and hence a deadlock occurred.

3 Mining Sequential Patterns for Counterexample Explanation

As we have seen above, in an interleaved trace of concurrent events the events belonging to a sequence which reveals an unforeseen interleaving do not necessarily occur next to each other. To the contrary, they can occur at an arbitrary, unbounded distance from each other. It therefore seems an obvious choice to use sequence or sequential pattern mining algorithms [1] [8] in order to devise error explaining subsequences of concurrent system executions. However, as we will argue in this section, this at first sight promising tool fails due to the inherent complexity of the problem.

3.1 Sequential Pattern Mining

We first define a subsequence relationship amongst sequences.

Definition 1. *A sequence* $\eta = \langle a_0, a_1, a_2, ..., a_m \rangle$ *is a* subsequence *of another sequence* $\rho = \langle \alpha_0, \alpha_1, \alpha_2, ..., \alpha_n \rangle$*, which is denoted by* $\eta \sqsubseteq \rho$*, if there exists integers* $0 \leq i_0 < i_1 < i_2 < i_3 ... < i_m \leq n$ *where* $a_0 = \alpha_{i_0}, a_1 = \alpha_{i_1}, ..., a_m = \alpha_{i_m}$ *[17].*

When applying a sequential pattern mining algorithm we consider a dataset of sequences, S, and a user defined threshold to decide whether a subsequence is frequent or not. The support of a sequence α is defined as the number of the sequences of S that α is a subsequence of:

Definition 2. $support_s(\alpha) = |\{s | s \in S \wedge \alpha \sqsubseteq s\}|$.

The sequence α is considered a sequential pattern or a frequent subsequence if its support is above a user defined threshold: $support_s(\alpha) \geq threshold$.

By contrasting the sequential patterns of the bad and the good datasets, we can extract patterns that are only frequent in the bad dataset. These patterns that are only frequent or common in the bad dataset, reveal anomalies, and hence can be indicative to the cause of the occurrence of deadlock,

$$\begin{aligned} anomalies = {} & \textit{sequential patterns} \text{ of the bad dataset} \backslash \\ & \textit{sequential patterns} \text{ of the good dataset} \end{aligned} \tag{1}$$

3.2 Challenges in Applying Sequential Pattern Mining Algorithms

In general, it can be shown that the problem of mining sequential patterns from a dataset of sequences is NP-hard. The complete proof is given in [24], [25]. The proof uses the following essential premises and lemmas:

1. In order to show that the *sequential pattern mining* problem is NP-hard it is sufficient to prove that the *frequent itemsets mining* problem [9], which is the problem of mining frequent *itemsets* from a dataset of *transactions*, is NP-hard. This is because the latter problem can be reduced to the former one. In the frequent itemsets mining problem, transactions are sets of items. An *itemset*, which is also a set of items, is frequent if the number of the transactions of which the itemset is a subset is above a user defined threshold.
2. In the frequent itemsets mining problem, the dataset of transactions can be represented as a *bipartite graph* $G = (U, V, E)$. U and V, which are the two distinct vertex sets of G, correspond to the set of items and the set of transactions, respectively. The edge set $E = \{(u, v) | u \in U \text{ and } v \in V\}$ of G represents all the (item, transaction) pairs.
3. The problem of enumerating all maximal frequent itemsets from a dataset of transactions corresponds to the task of enumerating all maximal *bipartite cliques* in a bipartite graph. A bipartite clique is a compelete bipartite subgraph of a bipartite graph.
4. Determining the number of maximal bipartite cliques in a bipartite graph is a *#P-complete* problem [21]. #P-completeness is used to capture the notion of the hardest counting problems, just as the concept of NP-completeness characterizes the hardest decision problems.

The above complexity arguments are based on worst-case complexity considerations [24]. A number of sequential pattern mining algorithms have been developed that have proven to be efficient in practice with respect to various test datasets [1], [23], [19]. However, the datasets that these algorithms have been evaluated on are sparse, with an average sequence length of less than 100. The densest dataset that an efficient sequential pattern mining algorithm, BIDE, can mine with a high support threshold of 90% has an average sequence length of 258 [23].

The characteristics of the bad and the good datasets of a number of Promela modeling case studies of concurrent systems are given in Table 1. In this table, the first four case studies are taken from [20]. The POTS model was developed by us as a sample model with numerous deadlock problems. This model is a non-trivial example of a telephony switch which comprises four concurrently executing proctypes 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. In Table 1, the column "#seq." gives the number of the sequences in the bad and the good datasets and the columns "avg. seq. len." and "max seq. len." represent average and maximum sequence lengths in these datasets, respectively.

Table 1. Dataset characteristics

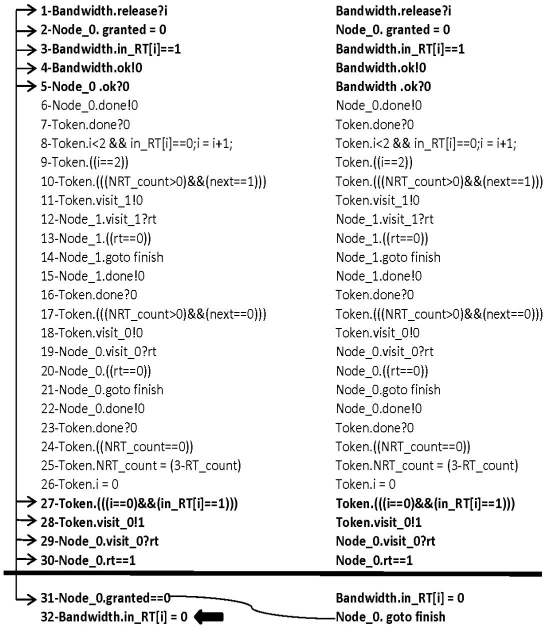
Model	#seq.		avg. seq. len.		max seq. len.	
	bad ds.	good ds.	bad ds.	good ds.	bad ds.	good ds.
Brp	660	25671	5985	10539	5580	10501
Rether	1061	26249	73263	134629	63201	134629
lann	989	20838	5737	12612	6369	12617
gear	614	10174	1994	4512	3837	4547
POTS	4109	11316	2995	7977	6134	6736

It can be inferred from Table 1 that the bad and the good datasets are highly dense with the average sequence length of more than 1000. We conclude that mining sequential patterns from the dataset of counterexamples generated from typical concurrent system models is intractable due to lengthy sequences and dense datasets.

4 Counterexample Explanation Method

To address the complexity challenges we encountered in mining sequential patterns from the bad and the good datasets, we abandon the feature of arbitrary distance between the events of a subsequence that we consider to reveal anomalies pointing at the causes for the occurrence of a deadlock. As an approximation we extract sequences that consist of consecutive events. These sequences are, in fact, substrings of the execution traces contained in the good and bad datasets.

Even though, as we have seen in the example of Sect. 2, a sequence that explains how a deadlock occurs is not necessarily the substring of a counterexample, it may contain portions which actually occur as substrings of a counterexample. In the example of Sect. 2, the sequences $\langle 1, 2, 3, 4, 5\rangle$ and $\langle 27, 28, 29, 30, 31\rangle$, which are portions of the identified subsequence for explaining the occurrence of a deadlock, are substrings of the counterexample. As we will explain in this Section, by extracting substrings from the counterexamples we can reveal parts of the sequences that give hints at why a deadlock occurs.



1-Bandwidth.release?i	**Bandwidth.release?i**
2-Node_0. granted = 0	**Node_0. granted = 0**
3-Bandwidth.in_RT[i]==1	**Bandwidth.in_RT[i]==1**
4-Bandwidth.ok!0	**Bandwidth.ok!0**
5-Node_0 .ok?0	**Bandwidth .ok?0**
6-Node_0.done!0	Node_0.done!0
7-Token.done?0	Token.done?0
8-Token.i<2 && in_RT[i]==0;i = i+1;	Token.i<2 && in_RT[i]==0;i = i+1;
9-Token.((i==2))	Token.((i==2))
10-Token.(((NRT_count>0)&&(next==1)))	Token.(((NRT_count>0)&&(next==1)))
11-Token.visit_1!0	Token.visit_1!0
12-Node_1.visit_1?rt	Node_1.visit_1?rt
13-Node_1.((rt==0))	Node_1.((rt==0))
14-Node_1.goto finish	Node_1.goto finish
15-Node_1.done!0	Node_1.done!0
16-Token.done?0	Token.done?0
17-Token.(((NRT_count>0)&&(next==0)))	Token.(((NRT_count>0)&&(next==0)))
18-Token.visit_0!0	Token.visit_0!0
19-Node_0.visit_0?rt	Node_0.visit_0?rt
20-Node_0.((rt==0))	Node_0.((rt==0))
21-Node_0.goto finish	Node_0.goto finish
22-Node_0.done!0	Node_0.done!0
23-Token.done?0	Token.done?0
24-Token.((NRT_count==0))	Token.((NRT_count==0))
25-Token.NRT_count = (3-RT_count)	Token.NRT_count = (3-RT_count)
26-Token.i = 0	Token.i = 0
27-Token.(((i==0)&&(in_RT[i]==1)))	**Token.(((i==0)&&(in_RT[i]==1)))**
28-Token.visit_0!1	**Token.visit_0!1**
29-Node_0.visit_0?rt	**Node_0.visit_0?rt**
30-Node_0.rt==1	**Node_0.rt==1**
31-Node_0.granted==0	**Bandwidth.in_RT[i] = 0**
32-Bandwidth.in_RT[i] = 0	**Node_0. goto finish**

Fig. 3. Part of a counterexample on the left, its corresponding part from a good trace on the right

The basis of our method is that we extract the common substrings of length l from the bad dataset and contrast them with those of the good dataset in order to reveal anomalies that explain the occurrence of deadlocks,

$$\begin{aligned} \textit{anomalies} = \textit{substrings of length } l \text{ of the bad dataset} \backslash \\ \textit{substrings of length } l \text{ of the good dataset} \end{aligned} \tag{2}$$

The length of the substrings, l, which is the parameter of the method, can take various values. Since substrings of length l can be extracted from a sequence of length n in $O(n)$ time, we avoid scalability problems. As we will see when presenting the experimental evaluation, the small value of $l = 2$ is adequate for explaining counterexamples using a fairly large set of case studies. To further justify this point, consider how a relatively short substring of length two can be indicative for the cause of a deadlock occurrence. In Fig. 3, the counterexample of Fig. 1 is given along with a non-failing trace on the right hand side. The given traces in this figure only differ in the last two events. The events above the horizontal black line are the same both in the counterexample and in the non-failing trace, and only the two events below the line are different. Therefore, the small substring $\langle 30, 31\rangle$ only occurs in the counterexample. Although $\langle 30, 31\rangle$ is only a small part of the spotted sequence which explains the occurrence of deadlock, $\langle 1, 2, 3, 4, 5, 27, 28, 29, 30, 31\rangle$, it can greatly help the model designer by using the knowledge about the functioning of the model to identify the other eight events of this anomalous subsequence in the counterexample. In particular, the substring $\langle 30, 31\rangle$ shows that the variables *Node_0.rt* and *Node_0.granted* have the values 1 and 0, respectively. The statements which affect the values of these two variables, can be easily found in the counterexample. The value of the variable *Node_0.granted* becomes 0 at step 2 and remains unchanged until the end of the trace. The value 1 of the variable *Node_0.rt* is due to the value 1 of the variable *in_RT[0]* while the value of this variable should be changed to 0 at the same step that the variable *Node_0.granted* gets the value 0.

Mainly based on what we have seen above, we assume that the substrings that only occur in the bad dataset, such as $\langle 30, 31\rangle$ in the above example, can aid the model designer to find the cause of the deadlock occurrence. The following subsections describe in detail the steps of our method.

4.1 Generation of the Good and the Bad Datasets

For generating the good and the bad datasets, we use the explicit state SPIN model checking tool [14]. The default search algorithm that SPIN uses for the exhaustive exploration of the state space is depth first search. When SPIN locates the first violating state, it stops the search and reports the path from the initial state to the violating state as a counterexample. The presence of one counterexample is sufficient to show that the model does not comply with the specification.

There is also an option in SPIN to not stop the search after locating the first violating state [14]. With this option, SPIN continues the search up to a given depth limit or until all states have been reached in order to locate all property violating states. Our current strategy for generating the bad dataset is to use this option of SPIN in order to explore the complete state space of the model and to detect all the violating states and their corresponding counterexamples. Since the default depth limit in SPIN is 10,000, we increase the depth limit until we can be certain that the complete state space has been explored. Since DFS is used by SPIN for exploring the state space, each violating state is visited once and so only one counterexample per violating state is generated.

Since the bad dataset contains the traces that violate some ϕ, the good dataset should include the traces that satisfy ϕ. Such traces can be generated by producing counterexamples to $\neg\phi$ because a counterexample that shows the violation of the negation of a property actually satisfies that property. This is justified by the following lemma:

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

Since the reachability property we consider in this paper is deadlock-freedom, we need to find a way to formalize the negation of that property in SPIN. Notice that while the absence of deadlock is a safety property, its negation, which claims the presence of deadlocks, is a liveness property. As a consequence, the counterexamples to the presence of deadlocks are *lasso-shaped* infinite traces [14].

We specify the presence of deadlock property in Promela, the modelling language of the SPIN model checker, by using a special state predicate named *timeout*. It becomes true when the system blocks, i. e., when no statement in the model is executed. We then specify the presence of deadlock property as *always eventually there will be a deadlock*, which can be expressed as requiring that always eventually the timeout predicate will become true. SPIN tries to generate a counterexample for this property. The resultant counterexample will be a lasso-shaped infinite trace that never deadlocks. For the generation of the good dataset we also use the SPIN option to not stop the search after generating the first counterexample for this property.

4.2 Contrasting Sequence Sets

Substrings of length l can be extracted from an execution trace by sliding a window of size l over it. Fig. 4 shows the nine possible substrings of length two that can be extracted from a trace of length 10 by sliding a window of size two over it. This set of substrings of length two, in fact, shows which two events occur next to each other in an execution trace.

Definition 3. *Sequence sets can be formally defined as follows: Let execution$(S) = \langle \alpha_1, ..., \alpha_n \rangle$, if the window is l actions wide, the set $P(S, l)$ of observed windows are the substrings of length l of S:*
$P(S, l) = \{w | w \text{ is a substring of } S \wedge |w| = l\}$ [6].

As an example consider $S = \langle abcabcdc \rangle$ and a window of size $l = 2$ slid over S. The resulting set of sequences of length two, P(S,2), will be: $P(S, 2) = \{ab, bc, ca, cd, dc\}$ [6].

The two following formulas define how to extract common substrings of length l from the bad and good datasets, respectively.

$$seq\_sets(bad, l) = \bigcup_{1 \leq i \leq n} \{P_i(S_i, l) | S_i \in bad, n = |bad|\} \tag{3}$$

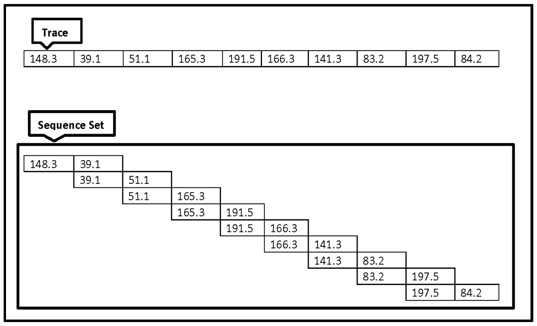


Fig. 4. Original trace with nine extracted short sequences of length two

$$seq\_sets(good, l) = \bigcup_{1 \le i \le m} \{P_i(S_i, l) | S_i \in good, m = |good|\} \tag{4}$$

The result set of the method, which is the set of substrings of length l that only occur in the bad dataset, is generated as follows:

$$anomalies = seq\_sets(bad, l) - seq\_sets(good, l) \tag{5}$$

The length of the short substrings, l, is the only parameter in computing the result set. We shall discuss the impact of choosing different values for l in the experimental results section.

Like all other debugging activities in which a *check*, *analyze*, *fix* loop is iterated until all the bugs are fixed, our method should also be used as an iterative process.

5 Experimental Results

In this section, we present a number of experiments in which we apply our counterexample explanation method to the Promela models of a number of concurrent systems which we took from [20]. The experiments were performed on a 2.67 GHz PC with 8 GB RAM and Windows 7 64-bit operating system. In fact, the experimental results illustrate how the outputs of our method can aid the user to identify the sequences of events in the counterexamples that explain the occurrence of a deadlock. We assume that the identified sequences directly allow the user to identify the faulty part of the model. This assumption is true for all case studies that we used as well as for the example presented in Sect. 2.

By using equation (5) we first generate anomalies which, as we explained above, consist of a set of substrings of length l that only occur in the bad dataset. We then take the following steps to facilitate the interpretation of the resulting anomalies for the user.

1. Since each substring can occur in multiple counterexamples, we extract for each substring the counterexample in which it occurs earlier than in other counterexamples. Each substring is only a portion of a sequence that explains the occurrence of a deadlock, so the model designer needs to identify other events of that anomalous sequence in the counterexample in order to understand how a deadlock occurs and consequently to localize the faulty part of the model. Intuitively, we assume that the substrings that occur earlier are closer to the beginning of such anomalous sequences in the counterexamples, and hence the user needs to inspect less events in order to identify them.
2. We rank the substrings based on their location in the extracted counterexamples in the previous step. Those that occur earlier in a counterexample will be ranked higher. It will be easier for the user to locate the sequence that explains the occurrence of a deadlock in counterexamples that are ranked higher than in those ranked lower.
3. Since multiple substrings can occur in a counterexample extracted in step one, for each counterexample we also list all the substrings that occur in it.

The output of the method will be a ranked list of tuples of the form $\langle \{ss_i | 0 \leq i \leq max\}, ce\rangle$ in which ss_i is a set of substrings generated by (5), max is the number of the substrings generated by (5) and ce is a counterexample containing the ss_is.

To evaluate the quality of the outputs generated by our method we propose a quantitative measure that enables us to compare different outputs. We define a score based on the amount of the effort that is required for locating a sequence that explains a deadlock occurrence in a counterexample by using the output of our method. Since these sequences directly allow the user to identify the faulty part of the model, as we assumed above, the computed score also reflects the amount of manual effort required for locating the faulty part of the model.

The output of our method consists of a number of substrings, so we first define a score for individual substrings. The score of the output will then be the score of the substring which is ranked first in the output. The score of a substring is defined based on the distance in terms of the number of the events between the location of the substring in a counterexample and the first event of a sequence that explains the occurrence of a deadlock in the same counterexample. This number, in fact, represents the maximum number of events that the user needs to inspect in the counterexample in order to find a sequence that explains the occurrence of a deadlock. Referring to the example of Sect. 2, the identification of such deadlock explaining sequences and their beginnings in the counterexamples is done manually by the user. Therefore, the score of a substring depends on the manually determined beginning of the deadlock explaining sequence. We normalize this distance with respect to the counterexample length. For the substring $\langle 30, 31\rangle$ in the example of Sect. 2, the score will be $\frac{30}{72}$, where 30 is the number of the events between the location of $\langle 30, 31\rangle$ and the start of the spotted sequence in the counterexample, and 72 is the length of the counterexample.

The following formulas define how a score is computed for an output of the method. In these formulas, *explanatory sequence* refers to a manually determined sequence that explains the occurrence of a deadlock. If $Loc_{\mathrm{ce}}(substring)$ and $Loc_{\mathrm{ce}}(explanatory\ sequence)$ represent the location of a substring and the location of the start event of an explanatory sequence in a counterexample, respectively, the distance between these two locations will be:

$$distance = Loc_{\mathrm{ce}}(substring) - Loc_{\mathrm{ce}}(explanatory\ sequence) \tag{6}$$

If $|counterexample|$ shows the length of the counterexample and $\mathrm{substring}_1$ shows the first substring in the output, the score of a substring and the output will be:

$$\begin{aligned} score_{\mathrm{substring}} &= 1 - \frac{distance}{|counterexample|} \\ score_{\mathrm{output}} &= score_{\mathrm{substring}_1} \end{aligned} \tag{7}$$

In Table 2, the results of applying our method to six case studies when $l = 2$ are given along with the corresponding scores. The name of the corresponding Promela file is given inside parentheses in front of the name of the model. The average running time of the method for these case studies is 52.44 sec. In this table, the last column shows the number of the root causes that can be detected by the model designer with the aid of the generated substrings of length two. As the numbers in this column show, with this method it is possible to detect multiple causes for the occurrence of deadlocks at the same time.

Table 2. Summary of the results of the method

Model	#$l = 2$ substrings	Score	#causes
Brp(brp.3.pm)	6	1	3
Modified Brp	6	1	2
Rether(rether.4.pm)	24	0.27	15
lann(lann.1.pm)	8	0.97	2
gear(gear.1.pm)	21	0.66	14
train-gate(train-gate.1.pm)	27	0.78	9

By increasing the value of parameter l, the number of the generated substrings will also be increased. Consequently, the model designer needs more effort for examining them. In Table 3, the number of the generated substrings for $l = 2$ and $l = 3$ for five case studies are given in the columns "#substrings $l = 2$" and "#substrings $l = 3$", respectively. The last column in this table shows the percentage of increase in the number of the generated substrings. We can see in this table that for the last three case studies, the number of the generated substrings of length three is significantly larger than those with length two. Therefore, the substrings of length three increase the amount of manual effort required for inspecting them. From Table 3 we can infer that substrings of length two impose less inspection effort on the model designer when analyzing

Table 3. Comparison of the number of the substrings with $l = 2$ and $l = 3$

Model	#substrings		rel. increase
	$l = 2$	$l = 3$	
Brp(brp.3.pm)	6	6	%0
Rether(rether.4.pm)	24	24	%0
lann(lann.1.pm)	8	29	%262.5
gear(gear.1.pm)	21	35	%66
train-gate(train-gate.1.pm)	29	62	%113.8

the counterexamples. As a consequence, the generation of substrings of length three only is done when no substrings of length two can be generated by (5).

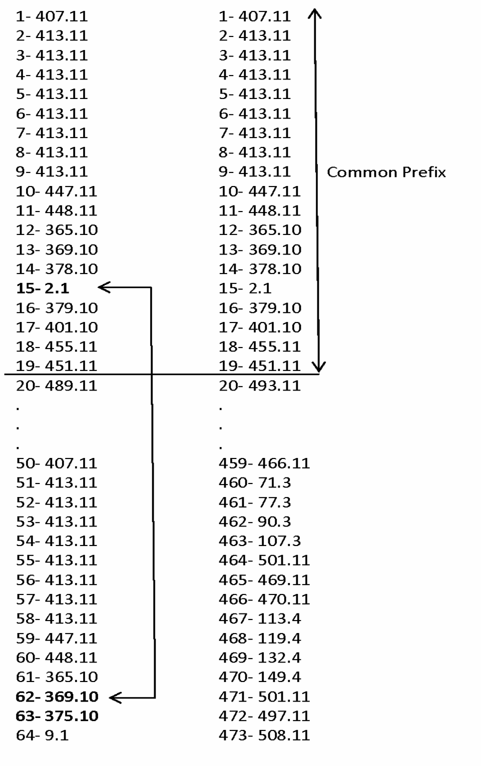
In Table 2, the Brp model has the highest score of 1 which means that the first ranked substring in the output coincides with the start of a sequence that explains a deadlock occurrence. Notice that we use the proposed method as part an iterative debugging process. After each run of the method, aided by the generated substrings the user will try to remove as many causes of deadlock as possible. In case the model still contains a deadlock after being modified, the user will apply the method again. This procedure can be iterated until all deadlocks in the model have been removed. As an example, after the first iteration on the Brp model the total number of counterexamples was reduced from 660 to 182 due to the removal of the root cause of some deadlock. The results achieved by applying the method to the modified version of the Brp model in the second iteration are given in the second row of Table 2.

6 Comparison With The Work by Groce and Visser

The most closely related work to ours is that of Groce and Visser [11]. It extends Java PathFinder with error explanation facilities. Given a counterexample, their method generates a set of *negatives*, which are multiple variations of that counterexample in which the error occurs, and a set of *positives*, which are variations in which the error does not occur. They analyze the common features of each set and the differences between the sets in order to provide an explanation for the counterexample. The focus of their work is on finite counterexamples demonstrating the violation of safety properties such as assertion violation and deadlock.

To compare our work with theirs we implemented the algorithm proposed in [11] for the generation of a set of positives for a given counterexample inside the Spinja [7] toolset. The main problem we encountered in applying this algorithm to our case studies was that we could not always generate a non-empty set of positives. This occurred, for instance, in our experiments with the Brp model. Notice that the potential emptiness of the positive set is also mentioned as a potential difficulty in practice in [11]. In our method, on the other hand, we consider the complete set of good traces that can be generated with the aid of

SPIN, and hence we cannot encounter the problem of an empty positive set for any case study that does at all reveal a "good" behavior.



1- 407.11	1- 407.11	
2- 413.11	2- 413.11	
3- 413.11	3- 413.11	
4- 413.11	4- 413.11	
5- 413.11	5- 413.11	
6- 413.11	6- 413.11	
7- 413.11	7- 413.11	
8- 413.11	8- 413.11	
9- 413.11	9- 413.11	Common Prefix
10- 447.11	10- 447.11	
11- 448.11	11- 448.11	
12- 365.10	12- 365.10	
13- 369.10	13- 369.10	
14- 378.10	14- 378.10	
15- 2.1	15- 2.1	
16- 379.10	16- 379.10	
17- 401.10	17- 401.10	
18- 455.11	18- 455.11	
19- 451.11	19- 451.11	
20- 489.11	20- 493.11	
.	.	
.	.	
.	.	
50- 407.11	459- 466.11	
51- 413.11	460- 71.3	
52- 413.11	461- 77.3	
53- 413.11	462- 90.3	
54- 413.11	463- 107.3	
55- 413.11	464- 501.11	
56- 413.11	465- 469.11	
57- 413.11	466- 470.11	
58- 413.11	467- 113.4	
59- 447.11	468- 119.4	
60- 448.11	469- 132.4	
61- 365.10	470- 149.4	
62- 369.10	471- 501.11	
63- 375.10	472- 497.11	
64- 9.1	473- 508.11	

Fig. 5. A negative and a positive in the Rether case study

The work in [11] proposes three different analyses for explaining counterexamples, namely *transition* analysis, *invariant* analysis and minimal *transformation* analysis between negatives and positives. Among these three analyses, only the third one, which takes the order of execution of actions into account, is similar to our method and can be used for revealing concurrency problems such as unforeseen interleavings. In this analysis, the authors of [11] compare a negative and a positive in order to determine the divergent sections of what they refer to as a state-action paths. These divergent sections along with the associated positive and negative form a *transformation.* In Fig. 5, a negative with 64 events along with a positive with 473 events derived for the Rether case study [20], are given. Due to space limitations, only the first 20 events and the last 15 events of these traces are shown in this figure. The first 19 events are identical both in the positive and in the negative, thus the divergent sections start from event 20 in both traces. These divergent sections last until the end of the positive and the negative since they do not share a common portion at the end of their traces. Therefore, the transformation generated by [11] will consist of two traces with 45 and 454 events. However, in our method two substrings of length two, $\langle 369.9, 375.9\rangle$ and $\langle 375.9, 9.0\rangle$, as well as the negative itself with 64 events are given to the model designer for further analysis. We conclude that while with the transformation analysis of [11] the model designer needs to inspect traces of 45 and 454 events, in our method the model designer needs to inspect at most 48 events in order to understand how a deadlock occurs. 48 is, in fact, the number of events between the location of $\langle 369.9, 375.9\rangle$ and the event "2.1" which is the beginning of the sequence that explains the occurrence of the deadlock in the trace. These two locations in the trace are 62 and 15, respectively, and in Fig. 5 they are connected by arrows and straight lines. In conclusion, our method appears, at least for the case study we considered here, to require less effort on behalf of the model designer in order to understand the reason for the occurrence of a deadlock than the equivalent analysis according to the work in [11].

7 Conclusion

We have presented an automated method for the explanation of model checking counterexamples demonstrating the occurrence of deadlocks in concurrent system models. In particular, we have focussed on deadlock detection using the SPIN model checker. By comparing a set of counterexamples with a set of correct traces that never deadlock, we extract a number of ordered sequences of actions that prove to point to the root cause of the deadlock occurrence in the model. Experimental results showed the effectiveness of our method and discussed measures to reduce the effort of the model designer when localizing the root cause for the occurrence of a deadlock in the model. We also compared our work extensively to related work, in particular the approach by Groce and Visser.

In future work we plan to reduce the computational effort that our method entails by generating subsets of good and bad traces based on some similarity measure. We also plan to extend our method to safety properties other than deadlock.

References

1. R. Agrawal and R. Srikant. Mining sequential patterns. In *11th International Conference on Data Engineering(ICDE'95)*, 1995.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, Cambridge, Massachusetts, 2008.
3. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2003.
4. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages, ACM*, 2002.

5. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *Proceedings of CAV 2009, LNCS. Springer*, 2009.
6. V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19thj European Conference on Object-Oriented Programming*, 2005.
7. M. de Jonge and T. Ruys. The SpinJa model checker. In *Model Checking Software, LNCS. Springer*, 2010.
8. G. Dong and J. Pei. *Sequence Data Mining.* Springer, 2007.
9. B. Goethals. Survey on frequent pattern mining, 2003. manuscript.
10. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2006.
11. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Model Checking Software, LNCS. Springer*, 2003.
12. J. Halpern and J. Pearl. Causes and explanations: A structural-model approach. part I: Causes. In *The British Journal for the Philosophy of Science*, 2005.
13. R. C. Holt. Some deadlock properties of computer systems. In *ACM Computing Surveys (CSUR)*, 1972.
14. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addision-Wesley, 2003.
15. D. Kroening, E. Clarke, and F. Lerd. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2004.
16. D. Lewis. *Counterfactuals.* Wiley-Blackwell, 2001.
17. D. Lo, S. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007.
18. S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi. Software fault localization using n-gram analysis. In *Wireless Algorithms, Systems, and Applications, LNCS. Springer*, 2008.
19. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *17th International Conference on Data Engineering (ICDE'01)*, 2001.
20. R. Pelanek. Benchmarks for explicit model checkers, 2006. http://anna.fi.muni.cz/models.
21. L. Valiant. *The Complexity of Computing the Permanent.* Theoretical Computer Science, 1979.
22. C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? causal analysis for counterexamples. In *Automated Technology for Verification and Analysis, LNCS. Springer*, 2006.
23. J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *20th International Conference on Data Engineering (ICDE'04)*, 2004.
24. G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.
25. G. Yang. Computational aspects of mining maximal frequent patterns. In *Theoretical Computer Science, Volume 362, Issues 1-3, Pages 63-85*, 2006.
26. A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, Burlington, MA, 2009.