Counterexample explanation in DiVinE model-checker

MASTER’S THESIS

Bc. Roman Plášil

Brno, 2011
Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: prof. RNDr. Luboš Brim, CSc.
Acknowledgement

I would like to thank my supervisor prof. RNDr. Luboš Brim, CSc. for guidance, discussions and the helpful advice. I am also grateful to the authors of the paper which was implemented in this work for providing further explanations on how the algorithm and JPF itself work. Last, but not least, I thank my family and closest friends for their support.
Abstract

The thesis focuses on counterexample explanation in model checking, which aims to provide useful information about the cause of an error so that the system designer can find and fix the error faster. We summarize some of the existing methods to deal with this problem and implement the method of Alex Groce and Willem Visser. This method describes three ways of comparing failing and unfailing runs to extract the cause of the problem. The original method was used in Java PathFinder, we implemented it in the parallel model checker DiVinE 2. The implemented method is evaluated on various models and a summary of the results is presented.

Keywords: parallel model checking, counterexample explanation, DiVinE, formal verification
# Contents

1. Introduction 1

2. Notions in Model Checking 2
   2.1. Explicit model checking with safety properties 2

3. A survey of approaches to counterexample explanation 5
   3.1. What Went Wrong: Explaining Counterexamples 5
   3.2. From Symptom to Cause: Localizing Errors in Counterexample Traces 10
   3.3. Error explanation with distance metrics 12
   3.4. Explaining Abstract Counterexamples 13
   3.5. Fate and free will in error traces 14

4. Our implementation in detail 16
   4.1. The environment 16
   4.2. The algorithms as implemented 19
   4.3. Implementation of the analyses 22
   4.4. Algorithm complexity 25

5. Algorithm Evaluation 27
   5.1. Outputs 27
   5.2. Summary 42
   5.3. Performance 46

A. Contents of the CD 50

B. Reading test reports 51
   B.1. Reading the data in clean Python 51
   B.2. Using the GUI 51
1. Introduction

In the past century, computers have changed our society in a significant way and now they are practically ubiquitous. They are used in a wide variety of areas, ranging from leisure activities to business and even to controlling critical systems such as airplanes. Requirements for reliability and safety in the latter group are naturally very high and thus we need formal methods of verifying our systems. One of such methods is called model checking. In model checking, the system is verified by an automatic program against a specification given either as a safety or a liveness property. A safety property states that “something bad never happens” and is usually expressed using propositional logic. A liveness property states that “something good keeps happening” and is expressed using a temporal logic such as CTL or LTL. When the model checker finishes its work, it either states that the property is fulfilled or reports a particular run of the system which violates the specification. This run is called counterexample and in the case of safety properties it has the form of a sequence of states of the system which leads to the state violating the safety specification.

When an error is found, it is up to the developer to find its cause and correct it. In this thesis, we call this problem counterexample explanation, even though not every published approach works specifically with the counterexample. The task of finding the specific cause of the error is not always easy for the system designer, particularly in larger systems, where they are confronted with long counterexample traces consisting of many states and actions. And while model checking helps significantly with finding counterexamples, there still is room for automated methods, which would help with localizing the error. As opposed to model checking, counterexample explanation is not a clearly defined problem, because what the designer needs is help with finding the specific cause, but it’s not clear what the help should be like or what actually a “specific cause of an error” is. There are a few points of view in publications on counterexample explanation and they are briefly summarized in Chapter 3.

In this work, we implemented the method of A. Groce and W. Visser in the parallel explicit model checker DiVinE 2.
2. Notions in Model Checking

This chapter introduces some basic definitions from model checking, which we will need in the later parts of this work.

2.1. Explicit model checking with safety properties

A safety property states that something bad never happens. For example that the cat and dog are never in the yard at the same time \[12\] or that in every state of the system, there is always at least one successor (i.e. there is no deadlock). These properties are usually expressed using propositional logic. This work is aimed only on safety properties as the algorithm from \[11\], which it implements is not designed for liveness properties.

In order to verify a system, we must construct its model. A model describes the behaviour of the system using some formalism. The original source code can be used in some cases for software systems, but often an abstraction needs to be employed to reduce the complexity or the system must be rewritten in a language the model checker supports.

An explicit model checking algorithm works with concrete states of the model under verification as opposed to symbolic model checking which operates over sets of states. Symbolic MC therefore can process models containing more states because the states in one group can be processed at once. To group the states together, Ordered Binary Decision Diagrams (OBDD) \[2\] or other formalisms can be used. However, building an optimal OBDD is not simple and in general working with symbolic representation is more complex than explicit representation. The formalisms in explicit model checking are straightforward but at the price of higher memory and CPU requirements. This work focuses on explicit model checking, because DiVinE, the model checker in which we implemented the described algorithm, is also explicit.

Explicit model checkers work over a graph structure containing states of the system and transitions between them. Formally, they work over a Kripke structure:

**Definition 1.** A Kripke structure is a tuple \((S, T, S_0, L)\) where

- \(S\) is a finite set of states
2. Notions in Model Checking

- \( T \) is a finite set of transitions (\( \forall \alpha \in T : \alpha \subseteq S \times S \))
- \( S_0 \in S \) is an initial state
- \( L : S \rightarrow 2^{AP} \) is a labelling function where \( AP \) is a set of atomic propositions

\((s, t) \in \alpha \) for some \( \alpha \in T \) can also be written as \( s \xrightarrow{\alpha} t \). When the transition name is omitted, as in \( s \xrightarrow{} t \), the formula \( \exists \alpha \in T : s \xrightarrow{\alpha} t \) is meant.

Counterexample explanation needs more information about the internal structure of a state than the model checker. Some of the information can be provided by the labelling function \( L \), but often the set \( S \) itself needs to be structured further. In practical systems, each state of a system comprises one control location for every process, thread or HW component and data valuation of all variables in the system. This means that the set \( S \) of states actually has a structure:

\[
S = C \times D
\]

To access the internals of a state \( s \) we can use the two projection functions \( c : S \rightarrow C \) and \( d : S \rightarrow D \).

To illustrate, consider the following picture of a Kripke structure. It consists of 4 states, \( s_1, \ldots, s_4 \) and transitions \( \alpha, \beta, \gamma, \delta \). Each state has three components, the first is the control location of the first process and it can be either \( a \) or \( b \), the second component is the control location of the second process with possible values being \( x \) or \( y \) and the last is an integer variable which we will refer to as \( z \).

![Kripke structure diagram](image)

Figure 2.1.: An example Kripke structure

The labelling function usually contains atomic propositions stating the values of variables so in our example, \( \{z = 0\} \subseteq s_1, \{z = 1\} \not\subseteq s_3 \).

Now we define the notion of a finite path, which is central to most of our discussion.
2. Notions in Model Checking

Definition 2. A finite path $\pi$ of Kripke structure $M = (S, T, S_0, L)$ is a finite sequence $s_0 s_1 \ldots s_n$ such that $n \geq 0, s_i \in S$ and $\forall 0 < i \leq n : s_{i-1} \xrightarrow{a} s_i$.

Definition 3. A finite path $t = s_0 \overset{a_1}{\rightarrow} s_1 \overset{a_2}{\rightarrow} \cdots \overset{a_k}{\rightarrow} s_k$ is a prefix of finite path $t' = s'_0 \overset{a'_1}{\rightarrow} s'_1 \overset{a'_2}{\rightarrow} \cdots \overset{a'k'}{\rightarrow} s'_{k'}!$ when $0 < k < k'$ and $\forall i < k : (i \geq 0 \Rightarrow s_i = s'_i) \land (i > 0 \Rightarrow a_i = a'_i)$.

A safety property $\phi$ holds in a Kripke structure $M$ if and only if $\phi$ holds in every state reachable from $S_0$. $\phi$ holds in a state $s \in S$, written as $s \models \phi$, when $\phi$ holds under valuation corresponding to $L(s)$. In our example, the formula $z < 3$ holds but $z = 1$ does not.

Given the Kripke structure and a formula, the model checker can verify it by visiting all states from the initial state using DFS or BFS and checking the validity of the formula at every state. When a state is found in which the formula does not hold, the search is terminated and the path from initial state to the failing state is presented to the user as a counterexample. This algorithm may also be called reachability, because it enumerates the reachable states and additionally performs the formula validity check.
3. A survey of approaches to counterexample explanation

This chapter provides an overview of some results in this relatively new field. Most of them are based on the intuitive idea that the difference between a correct and incorrect run should point to the actual cause of the error. Moreover, the correct run should be as similar to the counterexample as possible, to eliminate unrelated aspects of the system.

3.1. What Went Wrong: Explaining Counterexamples

Alex Groce and Willem Visser implemented an algorithm in the Java PathFinder model checker which explains violations of safety properties [11]. The algorithm finds a set of failing runs similar to the counterexample, called negatives, and a set of similar correct runs, which are called positives. The two sets are then analysed by one of three methods to find something in which negatives differ from positives and which should point to the cause of the error. This algorithm was implemented with some modifications in this work, so we present the algorithm here in greater detail in terms of the definitions given in Chapter 2.

In Java PathFinder a counterexample always ends in a virtual error state $s_k \in \prod$. Virtual states are not a part of the model, they are a helper notion used in JPF and the definitions. $\prod$ is the set of all virtual states. JPF therefore operates on the set of states given by

$$S = C \times D \cup \prod$$

Given a counterexample (which is a finite path) $t = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} s_k$, we define:

**Definition 4.** A negative (with respect to a particular $t$) is a finite path $t' = s_0 \xrightarrow{a'_1} s'_1 \xrightarrow{a'_2} \cdots \xrightarrow{a'_{k'}} s'_{k'}$ such that:

1. $c(s_{k-1}) = c(s'_{k'-1}) \land a_k = a'_{k'}$
2. \( s_k = s'_{k'} \)

Further, let \( \text{neg}(t) \) be the set of all negatives with respect to a counterexample \( t \).

**Definition 5.** A positive (with respect to \( t \)) is a finite path \( t' = s_0 \xrightarrow{\alpha'_1} s'_1 \cdots \xrightarrow{\alpha'_{k'}} s'_{k'} \) such that:

1. \( c(s_{k-1}) = c(s'_{k'-1}) \wedge \alpha_k = \alpha'_{k'} \)
2. \( s'_{k'} \not\in \prod \)
3. \( \forall t'' \in \text{neg}(t) : t' \) not a prefix of \( t'' \)

Again, let \( \text{pos}(t) \) be the set of all positives with respect to a counterexample \( t \).

Note that the last state of the counterexample, \( s_k \), belongs to the set of virtual error states \( \prod \), so we are in fact not comparing states of the system, only the kind of error. Similarly for \( \alpha_k \), the action leading to the virtual state – the action corresponds to checking an assertion, so \( \alpha_k = \alpha'_{k'} \) means that we are only interested in negatives which fail on the same assertion.

As the method works on a single counterexample, we will be omitting the parameter of \( \text{neg} \) and \( \text{pos} \).

We find only a subset of \( \text{neg} \) and a subset of \( \text{pos} \). No attempt is made to enumerate the sets \( \text{neg} \) or \( \text{pos} \) entirely as that could be computationally prohibitive and is not necessary. However, it also means that we are unable to fully check the condition that no positive is a prefix of a negative. It is sufficient for our purposes to check only the negatives that we generate.

The algorithm assumes that it can run a reachability procedure, which explores the state space from a given starting state and returns sets of positives and negatives. These are found by checking a safety property corresponding to the definitions; specifically, we are searching for a state which has the same control location as the state preceding the error state in the counterexample and which leads to the same error state (for negatives) or does not lead to an error state (for positives). Please note that the error state is actually a virtual state. The algorithm also requires that the reachability procedure can be limited to search only to a maximum depth and that it does not visit states that were already visited in preceding calls. Finally, the procedure must be able to find multiple states satisfying the condition given above and return a path to each of them.

The following pseudocode shows the basic version of negatives and positives searching algorithm. The intuitive idea is that it starts a reachability procedure called \( \text{MC} \) from every state on the counterexample with a limited search depth and collects the reported negatives and positives. In the pseudocode, \( \nu \) denotes the set of visited states, it is used and also filled by \( \text{MC} \) to avoid visiting states.
3. A survey of approaches to counterexample explanation

Repeatedly, \( n \) is the set of negatives found so far, \( p \) the set of positives and \( d \) is the maximum search depth parameter.

**Algorithm 1: BasicNegPos**

**Input:** counterexample \( t \), maximum search depth \( d \)

1. \( v := n := p := \emptyset \);
2. \( i := k - 1 \);
3. **while** \( i \geq 0 \) **do**
   4. \( (n, p, v) := MC(s_i, t, d, v) \);
   5. \( n := n \cup n \);
   6. \( p := p \cup p \);
   7. \( i := i - 1 \);
4. **end**
5. **for** \( t \in p \) **do**
6.   **for** \( t' \in n \) **do**
7.     **if** \( t \) is a prefix of \( t' \) **then**
8.       \( p := p - t \)
9. **end**
10. **end**
11. **end**
12. **return** \((n, p)\)

The publication proposes one more algorithm which, when integrated into the \( MC \) procedure, can find some more negatives and positives. It is triggered in state \( s \) when the depth limit is reached. The algorithm, named \( \text{Extension} \), tries to extend current search in such a way that the control locations and actions match some suffix of \( t \), the original counterexample. This means that \( d \) works like a kind of “edit distance” where the negative/positive is allowed to diverge from \( t \) for \( d \) steps, but then it must match \( t \) again.

In more detail, the algorithm first tries to find \( j \) such that \( c(s_j) = c(s) \), e.g. some state \( c_j \) on the counterexample whose control location matches that of \( s \) (the state where depth limit is reached). When this \( j \) is found, it further tries to find a path from \( s \) such that the actions and control locations along this path match those on the counterexample starting from \( s_j \).

3.1.1. Analyzing the runs

There are three methods described in the paper which are supposed to extract the essential information from the sets of negatives and positives. The first method interprets each run as a set of actions and compares these sets. The second method compares the invariants in positives with the invariants in the negatives.
3. A survey of approaches to counterexample explanation

Algorithm 2: Extension

Input: counterexample \( t \), starting state \( s \), set of already visited states \( v \)
Result: adds negatives and positives to the global sets

\[
\begin{align*}
1 & \text{ } j := i; \\
2 & \text{ while } j < k \text{ do} \\
3 & \quad \text{ if } c(s_j) = c(s) \text{ then} \\
4 & \qquad s' := s; \\
5 & \qquad l := j + 1; \\
6 & \qquad \text{ broken } := \text{ false}; \\
7 & \qquad \text{ while } l < k \land \lnot \text{ broken} \text{ do} \\
8 & \qquad \qquad \text{ if } \exists s'': s' \xrightarrow{a_i} s'' \land c(s'') = c(s_l) \land s'' \notin v \text{ then} \\
9 & \qquad \qquad \qquad s' := s''; \\
10 & \qquad \qquad \text{ else} \\
11 & \qquad \qquad \qquad \text{ broken } := \text{ true}; \\
12 & \qquad \qquad \text{ end} \\
13 & \qquad \qquad l := l + 1; \\
14 & \qquad \text{ end} \\
15 & \quad \text{ if } \lnot \text{ broken} \text{ then} \\
16 & \qquad \text{ if } s' \xrightarrow{a_k} s'' \text{ then} \\
17 & \qquad \quad \text{ if } s'' \in \prod \text{ then} \\
18 & \qquad \quad \quad \text{ add transition sequence to } s'' \text{ to negatives;} \\
19 & \qquad \quad \text{ else} \\
20 & \qquad \quad \quad \text{ add transition sequence to } s'' \text{ to positives;} \\
21 & \qquad \text{ end} \\
22 & \text{ end} \\
23 & j := j + 1; \\
24 & \text{ end} \\
\end{align*}
\]
Finally, we can try to convert a positive run into a negative one to find a change which could be the cause of the error.

**Transition analysis**

When some action appears in all negatives, but is missing from positives, it is very likely the cause of the error. Other properties, such as an action appearing in all positives can be useful too. That’s the intuition behind transition analysis. We build several sets of transitions and present them to the user as a possible indication of the error.

In transition comparison, we ignore the data component. We say that a finite path contains \( (c, \alpha) \) iff \( \exists n < k : c(s_n) = c \land \alpha_{n+1} = \alpha \). The sets we calculate are given in the following table:

<table>
<thead>
<tr>
<th>Set</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>trans(neg)</td>
<td>{ (c, \alpha)</td>
</tr>
<tr>
<td>trans(pos)</td>
<td>{ (c, \alpha)</td>
</tr>
<tr>
<td>all(neg)</td>
<td>{ (c, \alpha)</td>
</tr>
<tr>
<td>all(pos)</td>
<td>{ (c, \alpha)</td>
</tr>
<tr>
<td>only(neg)</td>
<td>trans(neg) \ \ trans(pos)</td>
</tr>
<tr>
<td>only(pos)</td>
<td>trans(pos) \ \ trans(neg)</td>
</tr>
<tr>
<td>cause(neg)</td>
<td>all(neg) \ \ only(neg)</td>
</tr>
<tr>
<td>cause(pos)</td>
<td>all(pos) \ \ only(pos)</td>
</tr>
</tbody>
</table>

Figure 3.1.: Transition analysis groups

The sets trans(neg), trans(pos) are only intermediate results and are too large, so they are not reported to the user. The remaining sets are of greater interest though. all( . . . ) contains all the transitions which occur in all negatives or positives respectively. only( . . . ) are the transitions appearing only in negatives or only in positives. When this information is still too general, cause( . . . ) provides a further reduction and when it’s not empty, it can provide precise location of the common behaviour, which differentiates the positive and negative sets.


**Invariant analysis**

For some programs, transition analysis based on the control flow of the program is unsuitable. Transition analysis ignores the data components of the runs, but in
3. A survey of approaches to counterexample explanation

some cases the data may be the discriminant between negatives and positives. According to the authors, applying transition analysis to d(s) instead of c(s) faces the problem that only some variables may be relevant rather than the full valuation.

Instead, data invariants in the negative runs are compared to invariants in the positive runs. The user chooses certain points in the program where the data invariants over negatives and over positives are calculated. These are then compared and presented to the user.

As before, please refer to the original paper for an example of this approach.

Transformation of positives into negatives

The last method finds a minimal transformation between every pair in pos × neg and sorts them according to some metric of transformation size. This results in a sequence of increasingly complex ways to cause a positive to fail.

The definition of a transformation used in the paper is simple, it only considers cases where both paths share a common prefix and a common control suffix. This corresponds to the Extension algorithm, which finds such traces. First, we define some helper notions before we define the transformation itself. The largest prefix of a finite path t is the prefix p of t that maximizes |p|, which is t without the last action and state. The largest prefix of a set of finite paths T is the finite path that is a prefix of all elements of T with the largest length.

Finally, a transformation of a positive t = s_o \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} s_k into a negative t' = s'_o \xrightarrow{a'_1} s'_1 \xrightarrow{a'_2} \cdots \xrightarrow{a'_k} s'_k is a pair of shorter finite paths p, u such that:

1. p is a prefix of both t and t’
2. u is a control suffix of both the largest prefix of t and the largest prefix of t’.

By definition of negative and positive, the final states of t and t’ do not share a control location. That’s the reason we work with the largest prefix. When the transformations are found, they can be additionally processed by the transition analysis algorithm. This has the advantage that it works on the essential parts of the runs.

Again, please refer to the paper for more information and an example.

3.2. From Symptom to Cause: Localizing Errors in Counterexample Traces

This paper [3] describes an approach which is very similar to the previous one. The authors implemented the method in SLAM [4], which is a tool that can
3. A survey of approaches to counterexample explanation

model check device drivers written in C and has been successful in finding errors in real drivers. It uses abstraction to reduce state space to finite and manageable sizes and Counterexample Guided Abstraction Refinement (CEGAR) to refine the abstraction when the counterexample produced is invalid. Nevertheless, the method for error localization from the paper is described on explicit state spaces. The authors also describe how procedure calls are handled in generating the state space and how it affects their algorithm, but we won’t go into such detail in this work.

The high level overview of the method is that when a safety violation is found, a counterexample is generated as usual, then a set of so called “correct” transitions is found and finally the cause is reported as the set difference between the transitions on the counterexample without the “correct” transitions. Then the control flow graph of the verified model is modified so as to remove the incorrect transitions. The whole process is repeated until no more causes are found.

The set of “correct” transitions is found by an algorithm which is in principle the same as reverse reachability. If we denote the erroneous state as $v$, the process is started from the states which have the same location as $v$ but don’t have an error. More specifically, a working set is established and initially it contains the correct states with the same location as $v$. Then for each state from the working set, the states that have a transition into the current one are added to the working set and the transitions are collected as the “correct” transitions. The algorithm ends when the working set is empty, that is when all backwardly reachable states have been enumerated.

The cause of the error is identified as

$$\text{project}(T) \setminus \text{project}(C)$$

where $T$ is the set of transitions on the counterexample, $C$ is the set of “correct” transitions and the function $\text{project}$ removes the variable valuations of the states of the transition and returns just a pair of locations. This is very similar to $\text{only}(\neg)$ from the previous paper.

The authors evaluated the method on Windows device drivers and checked for two properties. They received a total of 15 error traces in 8 drivers. In 11 of the traces the error was correctly localized, in 3 cases a change to the set difference formula was required and the last error wasn’t localized because of the use of abstraction.
3. A survey of approaches to counterexample explanation

3.3. Error explanation with distance metrics

The solution described in this article \cite{10} is implemented in the C Bounded Model Checker (CBMC). It works over C program code and instead of building an explicit state space it represents the program as a formula of propositional logic and uses a SAT solver to find a counterexample.

To approach counterexample explanation the authors use ideas of philosophers such as David Lewis. It is agreed upon by them that to explain something means to find a cause for it. Furthermore, Lewis says that a cause is something that distinguishes similar worlds containing the effect from worlds lacking the effect. More exactly, an effect $e$ has a cause $c$ at a world $w$ if and only if at all worlds most similar to $w$ in which $\neg c$ it is also the case that $\neg e$. An important notion is that we consider only the most similar worlds, because we don’t want to include worlds in which a different cause $c_2$ also causes the same effect $e$. The metric used to determine which worlds are the closest is hard to define for real worlds, but fortunately the situation is easier for program executions. The article defines one such distance metric and shows how it is used in an extension of CBMC for counterexample explanation.

CBMC is a bounded model checker which in practice means that it only considers a finite number of loop executions (this number is called unwinding depth). That is required in order for the model checker to be able to represent program executions as a propositional formula, called Static Single Assignment. In short, this representation works by creating a formula variable every time an assignment (even repeated assignment to the same C variable) or an if statement is found in the C program. From the values of these formula variables we can see what the values of the C variables were during the execution. The formula is joined with the negation of the specification and inserted into a SAT solver, which finds a counterexample. This representation has the advantage that there is no need to perform any alignment of the traces, on the other hand there is a disadvantage that both branches of an if statement are always considered in the distance metric even though only one branch is executed in every trace.

The distance metric is defined only for runs consisting of the same number of variables, which means that the same program and the same unwinding depth must be used. When this condition is satisfied, the distance $d(a, b)$ between runs is simply the number of differing variables. The distance metric is then used by the tool to find a correct trace whose distance to the counterexample is the smallest possible. To do this, the original formula along with an optimization target and the unnegated specification is entered into the pseudo-Boolean solver PBS \cite{11}. The goal is to find an execution that differs the least, so the optimization target is to minimize the number of differences. The advantage of this approach
is that we get the true most similar correct execution.

There is one more step before the differences are presented to the user. This step is called $\Delta$ slicing and it removes the differences which are not relevant to the error. A typical example is filtering out input values as they are always reflected in the program variables. $\Delta$ slicing is a variant of static or dynamic slicing, which is an approach which tries to find which variables influenced a certain program point. Slicing is already extensively described in literature. In this instance slicing can benefit from the use of a model checker, making the list of causes even smaller and more focused. To find the $\Delta$ slice, the pseudo-Boolean solver is used to solve an optimization problem where the distance metric is the same, but the formula is modified so that a smaller slice is found. The new solution may not correspond to a possible run of the program, it is only used to reduce the size of the explanation. Please see the original paper for a more detailed description.

There was also an attempt to perform the search for a most similar correct run and the $\Delta$ slicing in a single invocation of the pseudo-Boolean solver, but that turned out to be counterproductive. The main problem was that without already having a correct run, the slicing part cannot optimize against one specific run.

As for evaluation, the authors modelled the Traffic Collision Avoidance System (TCAS) from [8]. In some cases the explanation algorithm worked well, but they also encountered a problem where simply a different input was chosen to produce a correct run, which was unhelpful in resolving the error. After running $\Delta$ slicing to the explanation, almost nothing was left, which means that $\Delta$ slicing was useful in that it removed the unhelpful differences. The scores table in the article also contains a comparison with explain implemented in JPF and in this work, but due to differences between the two systems the comparison is of somewhat dubious value according to the authors.

### 3.4. Explaining Abstract Counterexamples

This paper [5] is an extension of the previous algorithm. The method described here is implemented in the tool MAGIC. The basic scheme of operation is the same as in the previous article, but in this case it works over abstract state spaces and abstract counterexamples. Instead of concrete values of variables, the model checker works with sets of predicates on variable values, so an abstract state may describe a set of concrete states. Similarly an abstract counterexample is a list of predicates. Abstractions are generated automatically and refined if needed (CEGAR). As opposed to the previous approach, loop unrolling in bounded model checking is used only in the explanation phase, because the counterexample is
A survey of approaches to counterexample explanation

found using other means. Furthermore, Static Single Assignment is not used, which means that we need to align the (abstract) states, but on the other hand there are no values from paths which were not executed. Using predicates instead of concrete values has the advantage that the users work with high-level concepts which hopefully match their intuition about the program as opposed to concrete values for which they have to build an abstraction first.

To define a distance metric, we first need to align the two executions. The alignment is expressed using a logic formula and requires that only states with matching control locations are aligned. The other properties of the alignment are intuitive; a state has at most one aligned state, ordering is preserved and gaps are allowed. The formulas for the distance metric are rather involved, but the idea is that it sums the number of mismatched predicates and actions when the states are aligned with the number of unaligned states. The distance between two runs is then the minimum of the sum over all possible alignments.

To find the closest correct execution, again a formula is generated from the program under verification, then the constraints on alignment and the $\Delta$ variables expressing the difference between the original counterexample and the searched correct execution are added to it. This is then processed by a pseudo-Boolean solver which minimizes the distance metric, which means that it finds a correct run with as few differences as possible together with an alignment. Furthermore, the execution found must be checked that it is not spurious as it operates in an abstracted environment. If it is spurious, a clause is added to the formula which forces generating a different trace and the process is repeated.

Results of comparing this newer method with the previous one are mixed. On one hand, abstract model checking has the advantage that it allows to process larger systems and that, at least intuitively, predicates provide more information to the user than concrete values. There was success on some programs and on some kinds of errors. On the other hand, tests of the new method on TCAS or $\mu$C/OS-II 2.0 failed to show an improvement.

3.5. Fate and free will in error traces

We will present this article only briefly. Situated in HW model checking, the idea lies in marking segments of the counterexample as either “fated” or “free”. In fated segments, it’s inevitable for the system to reach a state that leads to an error, whereas in free segments the system could avoid the error, but makes a mistake. In order to mark the counterexample, the user must first divide input signals into two groups, “controlling” and “noncontrolling”. The first group contains signals that control the circuit and represent a “hostile” environment while the other
3. A survey of approaches to counterexample explanation

group contains the data signals. In verification of parallel software, the decisions of the process scheduler can be considered as controlling signals.

The run of the system can be thought of as a two player game where the environment is trying to steer the system into an error. At every state, the environment chooses values of the controlling variables and then the system chooses values of the noncontrolling variables. The system then moves according to its transition relation. The model checker finds sets $L_i$ of states where states in $L_i$ are exactly those where the system must make $i$ mistakes to be forced to an error. Using these sets, a counterexample containing the least possible free segments can be found and presented to the user for analysis.
4. Our implementation in detail

4.1. The environment

We implemented the method in DiVinE 2, an explicit parallel model checker developed at the Faculty of Informatics, Masaryk University. It can use several state space generators and even external modules, but this work supports only DVE, a language based on finite state automata designed for the first version of DiVinE. In theory, it could work with any other language that has a notion of processes, process locations and variables, but an interface for accessing this information would have to be designed first.

In the following we use only the name DiVinE, omitting the version number.

4.1.1. Model structure

The algorithm needs to access the internal structure of the states, such as processes, variables and locations. We briefly describe the structure of DVE, but recommend for an extended explanation.

A DVE model consists of global variables, channels and processes. Each process can have local variables, locations and transitions. All processes run concurrently either synchronously (all processes move at the same time) or asynchronously (only one process moves at a time, unless synchronization between individual processes is used). A process makes a step by performing one of currently enabled transitions leading out of its current location. Each transition can have a boolean expression called guard and a synchronization requirement called sync, which is a name of a channel and a read or write operation. The transition is enabled when its guard evaluates to true and it is possible to perform the synchronization operation. When a transition is taken, variables are updated according to effects given on the transition.

```plaintext
int x = 0;

process A {
    int count;
    state q0, q1;
}
```
4. Our implementation in detail

```plaintext
init q0;
trans
    q0 -> q1 { guard x % 2 == 0; effect x = x + 1; },
    q1 -> q0 { effect count = (count + 1) % 10; };
}

process B {
    state z0, z1;
    init z0;
    trans
        z0 -> z1 { guard x % 2 == 1; effect x = x - 1; },
        z1 -> z0 { };
}

system async;
```

In this example, written in the syntax of DVE, we have two processes, A and B, one global 32bit variable `x` and a variable `count` local to process A. Process A has two locations, `q0`, `q1`, similarly for B. The system is asynchronous.

A state in DVE consists of a valuation of global variables, contents of buffered channels, locations of all processes and values of all local variables in all processes. Variables can be either integers (8 or 32 bits wide) or one dimensional arrays. Channels in DVE can transfer multiple values at the same time, but we only consider buffered channels carrying one value at a time, because Daikon, the invariant detector used in invariant analysis, supports only one dimensional arrays, which correspond to single-value buffered channels.

A state in the example above can look like this:

- `x = 1`
- `location of A = q1`
- `A.count = 3`
- `location of B = z0`

As opposed to Java, DVE is not a real programming language, so working with our implementation of the explanation may not be as accessible to regular programmers as opposed to JPF, which operates over a well known language. That is a disadvantage, because the primary aim of counterexample explanation is to make using a model checker even easier. There have been attempts to build a new language for DiVinE, such as [13] (written in Czech), but as of time of writing, none have been completed.
4. Our implementation in detail

4.1.2. Types of safety errors

There are four types of errors that can occur in a state. A state which has some kind of error is called a goal.

**deadlock** The state has no successors. This happens when there is no enabled transition in any of the processes.

**assertion violation** DVE allows specifying an expression which must be true in a location of a process. A state has this kind of error when a location has an assigned assertion which evaluates to false.

**expression** A DVE expression given to DiVinE on the command line. The expression is evaluated in each state and the state is erroneous when it evaluates to false.

**error** Occurs when the DVE generator encounters an error such as two synchronized transitions accessing the same variable.

DiVinE allows selecting which kinds of goals we are interested in using command line switches.

There are no virtual states (see 3.1) in DiVinE, so the definition of negatives and positives is changed to express the same meaning using different concepts. For a negative, we require the following:

- \( c(s_k) = c(s'_k) \)
- type of error in \( s_k \) = type of error in \( s'_k \)
- (type of error in \( s_k \) = assertion violation) \( \Rightarrow \) (assertion violated in \( s_k \) = assertion violated in \( s'_k \)).

In these definitions, unprimed symbols refer to the counterexample and primed to the negative or positive, as usual. In our setting, \( s_k \) is the last “real” state of the system and it corresponds to \( s_{k-1} \) in JPF. We don’t consider the action \( a_k \) as in JPF, because, again, this action leads to the virtual state. Instead, we check that we end up with the same type of error and in the case of assertion violation, that we deal with the same assertion in both cases.

A positive must satisfy the following conditions:

- \( c(s_k) = c(s'_k) \)
- \( s'_k \) has no error
- \( \exists s'_{k'+1} \) such that \( s'_k \rightarrow s'_{k'+1} \), \( s'_{k'+1} \) has no error and \( s_{k+1} \) does not lie on a negative.
4. Our implementation in detail

Here we also omit the requirement on the last actions to be the same. In JPF, the actions mean checking deadlock or assertion violation, but there is no counterpart for that in DiVinE. The requirement that the positive is not a prefix of a negative is strengthened and helps filter out more positives that may end in an error.

However, as we don’t always find all negatives, we may still end up with a positive that can continue to an erroneous state. That may or may not be a problem, depending on the kind of model and the kind of error. For example in ex1.dve, which models the first example from [11], the second positive may continue and end up in an erroneous state. This has not been detected, because no negative passes through the last state of the positive.

The following three trails illustrate our requirements on positives and negatives. For a system with one process (whose control location is the first component) and one variable x (whose value is written in the second component) with an assertion that $x \neq 0$ in control location c, we may have the following counterexample:

\[(a, 0) \rightarrow (a, 1) \rightarrow (b, 1) \rightarrow (c, 0)\]

a negative may look like this:

\[(a, 0) \rightarrow (b, 0) \rightarrow (c, 0)\]

a positive may be:

\[(a, 0) \rightarrow (a, 1) \rightarrow (b, 1) \rightarrow (c, 1) \rightarrow (d, 1)\]

On the other hand, the following run which ends in a deadlock may not be a negative, because only runs with the same type of error as in the counterexample (assertion violation) are considered:

\[(a, 0) \rightarrow (a, 1) \rightarrow (b, 1) \rightarrow (c, 2)\]

4.2. The algorithms as implemented

To find the positives and negatives, we use two algorithms, BasicNegPos (alg. 1) and Extension (alg. 2). In our implementation, however, they are structured in a slightly different way. When iterating backwards over the counterexample, we run one step of the basic search and one step of the extension per state of
counterexample (see pseudocode below).

**Algorithm 3: Explain**

1. $t := \text{find\_counterexample}();$
2. $ps := ns := \emptyset;$
3. $i := k - 1;$
4. **while** $i \geq 0$ **do**
   5. $(p, n) := \text{explain\_step}(s_i);$  
   6. $ps := ps \cup p;$  
   7. $ns := ns \cup n;$  
   8. $(p, n) := \text{extension\_step}();$  
   9. $ps := ps \cup p;$  
   10. $ns := ns \cup n;$  
   11. $i := i - 1;$
5. **end**
6. $pos := \text{extract\_traces}(ps);$  
7. $neg := \text{extract\_traces}(ns);$  
8. $\text{transition\_analysis}(pos, neg);$  
9. $\text{invariant\_analysis}(pos, neg);$  
10. $\text{transformation\_analysis}(pos, neg);$  

The basic algorithm remains in the same form, but the extension algorithm is changed. As opposed to the original approach of calling the extension algorithm whenever the basic algorithm crosses the maximum depth, we call it only after one step of the basic algorithm is done. The reason for this is the parallel infrastructure of DiVinE, which is based on BFS and does not allow a nested DFS search easily. Instead, we remember all the visited states behind the depth barrier and run the extension for these states in a batch.

Both explain\_step() and extension\_step() return only the “seed” of negative or positive, the last edge of the trail. We find and store the entire trail in extract\_traces() using parent pointers after the entire search is done. This is a requirement of the parallel BFS algorithm, the trail is not immediately available as in the case of DFS. It also means that we get trails which are close to being the shortest paths from the initial state to the seed.

**explain\_step()**

This is a parallel BFS reachability starting from the state $s_i$ given as an argument. It checks the requirements for negative in each state and the requirements for positive in each edge. It marks states as visited to avoid seeing any state more than once. Thanks to this, all the calls to this function combined take time linear
4. Our implementation in detail

in the size of the graph, which is divided among available processors on the
machine (with some overhead). All states whose distance from $s_i$ exceeds the
depth limit are stored in an array for use in the next step.

extension_step()

Algorithm 4: extension_step

Input: the set $E$ of states behind the depth barrier, counterexample $t$
Input: maximum depth $d$
Result: adds negatives and positives to the global sets

1 foreach $i \in E$ do
2     Extension($i, d$);
3 end

This function runs the pseudocode for each state that was remembered in
explain_step(). Our implementation of Extension follows the pseudocode,
but it uses the parallel BFS infrastructure available in DiVinE. In each state, it
selects only one successor, using the CAS synchronization primitive. When it
is no longer possible to follow the chosen path, no attempt is done to retreat,
the current search ends and we try the next alignment (algorithm Extension
proceeds to the next iteration of the while loop on line 2). Time complexity
of this algorithm depends on the number of states in the set $E$, on the length
of counterexample $t$, on the number of matches of input state and the coun-
terexample and on the number of outgoing edges on each state visited. Let
$m_i, i \in E$ be the number of matches of state $i$ with the counterexample $t$. Next,
let $D_{i,k}, i \in E, k \in \{1, 2, \ldots, m_i\}$ be the sum of outdegrees of states visited when
processing state $i$ and match number $k$. Then the time complexity of all calls to
the function is

$$|t| \cdot \sum_{i \in E} \sum_{k \in \{1, 2, \ldots, m_i\}} D_{i,k}$$

A simpler upper bound is

$$|t| \cdot |t| \cdot d_m$$

which uses the fact that the number of steps made by Extension for one starting
state is never greater than the length of the counterexample times the maximal
degree of states along the path ($d_m$ is the maximal degree of the graph). This
part runs sequentially.
4. Implementation of the analyses

Implementing the analyses was rather straightforward.

4.3.1. Transition analysis

Before running this analysis, the program must first extract the transition information, because the trails are just lists of states. We work directly with the transition objects from the DVE parser and generator. One transition of the system may consist of one or two transitions of processes, the latter is the case when synchronization is used. The rest are just set operations. For comparison of transitions, we use only the locations of processes which move in the transitions, not the entire $c(s)$.

As an example, we’ll use one of the models we evaluated the method on. It is called rw.6.dve. This is a model of a readers/writers lock with an artificial modification used to demonstrate transition analysis. The writer process alternates the variable $X$ under lock between 1 and 0. Additionally, it keeps a local copy of the variable. As there is just one writer, the copy is correctly synchronized with the global variable. The error lies in not alternating the global value correctly while the copy is alternated. See Figure 4.1 for the automata.

![Figure 4.1.: The writer process.](image-url)
4. Our implementation in detail

This causes the global and local copies of the variable to get out of sync and that's where an assertion is violated. The output of transition analysis restricted to the writer process looks like this:

\begin{verbatim}
ALL: (pos) ----------------------------------
<0.0> / q0 -> wait { guard writer==0; effect writer = 1; }
<0.2> / wait -> CS { guard X==0; effect X = 1, X_copy = 1; }
<0.3> / CS -> q0 { effect writer = 0; }

ALL: (neg) ----------------------------------
<0.0> / q0 -> wait { guard writer==0; effect writer = 1; }
<0.1> / wait -> CS { guard X==1; effect X_copy = 0; }
<0.2> / wait -> CS { guard X==0; effect X = 1, X_copy = 1; }
<0.3> / CS -> q0 { effect writer = 0; }

ONLY: (pos) ----------------------------------

ONLY: (neg) ----------------------------------

CAUSE: (pos) ----------------------------------

CAUSE: (neg) ----------------------------------

<0.1> / wait -> CS { guard X==1; effect X_copy = 0; }
\end{verbatim}

We can see that the transition \texttt{wait -> CS} which forgot to set $X$ to the new value is present in \textit{all(neg)}, which means that it was found in all the negatives, it's also in \textit{only(neg)}, which means that it was not found in any of the positives. Both these facts result in the transition being present in \textit{cause(neg)}.

4.3.2. Invariant analysis

Like the original paper, we employed Daikon \cite{9}, the Java invariant detector, for this task. To use this analysis, the user must specify \textit{instrumentation points} in which the values are stored and later used to determine the invariants. In Java, this is done by inserting a call to the JPF into the verified program. DVE does not support anything like that, so we opted for using an expression to select the states to be instrumented. This allows good flexibility and does not need any changes to the DVE parser and generator, on the other hand it may not be as user friendly. To specify an instrumentation point, use the \texttt{-instr=<DVE expression>} command line option. There can be more than one such option in the command line.

Daikon accepts text files containing first the definitions of the variables and then the values. All variables are \texttt{int} or arrays thereof. Buffered channels are also translated into arrays. Daikon does not support two dimensional arrays, so we only support channels transmitting one value at a time. It is also possible to
4. Our implementation in detail

specify relations between the variables to avoid generating invariants comparing variables which have no semantic relation. As DiVinE does not have this information, we don’t use this feature of Daikon. According to an email from the authors of JPF, their tool let the user specify these relations by hand.

The following is the output of invariant analysis on our example model. Some lines were removed for brevity. We can see that the invariant that the local and global copies of X are equal is present in the positives, but not the negatives.

```plaintext
instrument0.neg:::POINT
X == 1
readers one of { 0, 1 }
writer one of { 0, 1 }
r one of { 0, 1 }
===========================================
instrument0.pos:::POINT
X == W_0.X_copy
X == 1
readers one of { 0, 1 }
writer one of { 0, 1 }
r one of { 0, 1 }
```

4.3.3. Transformation analysis

In this step, we generate transformations from negatives to positives, as described in the paper. We then sort them according to the size of transformation (length of the positive part + length of the negative part) and print the first three smallest transformations because printing all of them would produce too much output for the user to read. This filtering is not described in the original paper, we found out about it from discussion with the authors. Each transformation consists of a subtrail of a negative and a subtrail of a positive. The first state of both the positive and the negative is the same, for easier orientation. Similarly for the last state, where it’s just the locations which are the same.

Transformation analysis often works similarly to transition analysis. For the example 4.1 one of the transformations we get is the following:

Transformation size 4
positive part (from p0):
negative part (from n0):
[X:1, readers:0, writer:0, r:0]; W_0:[q0, X_copy:1]; R_0:[q0, local:0]
[X:1, readers:0, writer:1, r:0]; W_0:[wait, X_copy:1]; R_0:[q0, local:0]
4. Our implementation in detail

[X:1, readers:0, writer:1, r:0]; W_0:[CS, X_copy:0]; R_0:[q0, local:0]
[X:1, readers:0, writer:0, r:0]; W_0:[q0, X_copy:0]; R_0:[q0, local:0]

The positive part is empty, which means that the transformation suggests removing the subtrace presented as the negative part, which contains the erroneous transition.

4.4. Algorithm complexity

The total time complexity is a sum of complexities of all phases:

1. find_counterexample is $O(|S| + |T|)$ where $|T|$ is the number of transitions
2. All executions of explain_step combined are also in $O(|S| + |T|)$
3. All executions of extension_step combined are in $O(|S| \cdot |t|^2 \cdot d_m)$. Extension is run for each state behind the depth barrier, which can never be more than $|S|$ so multiplying that by the upper bound from the description of extension_step gives the worst case complexity. In most cases the complexity is significantly lower.
4. Extracting the traces is linear in the number and length of the traces. We’ll denote the maximum length of a negative or a positive as $L$. The program finds up to 50 positives and 50 negatives by default, which gives a worst case complexity $O(100 \cdot L)$.
5. Transition analysis runs set operations on negatives and positives, but the complexity is dominated by the sorting of all traces which may be quadratic at worst. We get $O(100 \cdot L^2)$.
6. Invariant analysis is performed by an external program.
7. Transformation analysis takes up to $O(50 \cdot 50 \cdot L)$ steps.

The overall time complexity of our program is dominated by the state space search, so we get

$$O\left(|T| + |S| \cdot |t|^2 \cdot d_m\right)$$

DiVinE2 is a parallel model checker, which can run algorithms both on multiple processors and multiple computers at the same time, communicating through shared memory in the first case and using MPI in the latter. The parallel infrastructure is shared by all algorithms and makes implementing new algorithms easier (provided that the parallel decomposition is the same). Our implementation supports shared memory parallelism in the basic search for positives and negatives so the most time demanding part of the algorithm can be accelerated when multiple processors are available. The extension algorithm is implemented as parallel, but it always selects just one successor, so for practical purposes it
4. Our implementation in detail

can be considered as sequential. The analyses of the traces are sequential as they are not computationally intensive. See 5.3 for actual run times on different kinds of models.

As for space complexity, no part of the program uses more than a constant amount of space per state, which gives space complexity linear in the number of states.
5. Algorithm Evaluation

We ran the algorithm on several models with several kinds of errors to evaluate its ability to help with locating the errors. Some of the models were taken from the BEEM [5] database, some of them are models from the Software-artifact Infrastructure Repository [8] rewritten in DVE and the rest of them were created during the evaluation.

This section contains the list of tested model instances along with a short description of the model, of the error and of the output of the explain algorithm. At the end of each evaluation, there is a one-word summary. It can be one of the words useless, touches, notbad and great, which represent four levels of usefulness for localising the error.

The complete information about all the runs, including the exact input and output, command line parameters and run time can be found on the attached CD. See Appendix B for information about how to use the data. Only the runs where just one worker is used are repeatable, with multiple parallel workers the code is not deterministic anymore and may find a different counterexample, positives and negatives.

At the end of the section, we provide a summarization of the results.

5.1. Outputs

5.1.1. lamport_47.244.dve

This is a DVE model of the Lamport’s Mutual Exclusion Protocol [15] taken from BEEM.

Error description:

After several steps, the model ends up in a deadlocked state because of the error. The problem lies on transition $q_2 \rightarrow q_{22}$ \{guard $y \neq 255$; effect $b[x]=0$;\} of every process. The correct effect is $b[i] = 0$ where $i$ is the index of the process. When $x$ is different from $i$, which happens when another process tries to enter the CS, the error occurs.
5. Algorithm Evaluation

Evaluation

Transition analysis  The analysis outputs many transitions and the known offender is buried among them.

Invariant analysis  Instrumentation points were chosen based on the knowledge of the error in the model, but were not specified overly accurately. There is a large amount of invariants, but none of them provides any useful insight.

Transformation analysis  All the first three smallest transformations are symmetric and propose removing certain transitions from the negative. Unfortunately, neither this analysis is helpful in finding the problem.
  Summary: touches

5.1.2. lamport_47.244.dve

This is another run of explain on the same model with the same error. This time we found a different counterexample and the analysis for it is more successful.

Evaluation

Transition analysis  This analysis outputs the one problematic transition in \(\text{only}(\neg)\), along with another transition. This is a good result, since \(\text{only}(\neg)\) is an important group and the signal to noise ratio in the group is good.

Invariant analysis  As before, there are too many invariants and the output is not helpful.

Transformation analysis  All the first three smallest transformation do in principle the same thing – delay the deadlock by counting in the state \(q4\). Unfortunately the positives here have an extension into a negative, so they are not really useful.
  Summary: notbad

5.1.3. msmie_13.65.dve

This is a model of a protocol for communication between processors in a real-time control system. The model for this protocol in BEEM has an error, which causes it to deadlock.
5. Algorithm Evaluation

Error description:

When studying the model itself and the location of the deadlock, it is rather easy to see where the error lies. The master process template is waiting in state change for the first element of \( b \) to become MASTER where clearly the intention is to wait for any element of the array to become MASTER. A similar error occurs in the transition from no_readers to change.

Evaluation

The algorithm finds no positives, likely because all candidates are filtered out, because they can continue into a negative. With no positives, none of the three analyses can be used, but at least we know that there must be a fundamental error which does not allow the existence of a positive.

Summary: useless

5.1.4. train-gate_53.277.dve

This is a train gate controller which controls which of the several trains can cross a bridge. The model comes from BEEM.

Error description:

There is a deadlock caused by incorrect translation of the system into DVE. This model uses a global variable \( e \) together with messages sent via global channels. The recipient of the message is stored in the variable \( e \). However, a race condition causes the variable to be out of sync.

Evaluation

Transition and transformation analysis  Both transition and transformation analyses suggest that it's good when the Train_7 starts approaching sooner than Train_1 and bad when it's vice versa. That is a correct hint, but it's still not easy to find the error just with this information.

Invariant analysis  We placed an instrumentation point at the end of all negatives. As usual, Daikon outputs a large number of invariants which is hard for a user to process. However we can see that in all negatives \( e = 7 \) (\( e \) is incorrectly represented as an array from which only the first element is used) and in all positives \( e = 1 \). This may be useful, because the error is a race condition on this variable.
5. Algorithm Evaluation

Summary: touches

5.1.5. rw_1.dve

An implementation of the readers/writers lock with readers preference. In this model variant, we have only one reader, to make the negatives and consequently error analysis simpler.

Error description:

The error lies in the reader not releasing the writer lock when it is done, causing a deadlock on the next operation.

Evaluation

Transition analysis  Cause(neg) contains the entire sequence of actions of a reader, which is not surprising, because that's the only way to reach the problematic transition. This information does not give a specific location of the error, but at least we know that the reader is responsible.

Invariant analysis  This type of analysis is not suitable for this problem, so we skipped it.

Transformation analysis  This output provides information very similar to transition analysis. It suggests completely removing reader's actions and replacing them either with a writer's move or just doing nothing.

Summary: touches

5.1.6. pushparent_1.dve

We created a model for a function which adds an element to a set represented by a fixed size array. The C code for the function looks like this:

```c
1 for ( size_t ix = 0; ix < NUM_PARENTS; ix ++ ) {
2    if ( parents[ ix ] == n ) return false;
3    if ( parents[ ix ].valid() ) continue;
4    parents[ ix ] = n;
5    return true;
6 }
```

The array should contain all elements inserted into it and no element should be repeated. This code is correct, but there is an artificial error in the model.
5. Algorithm Evaluation

Error description:
The first error is omitting the `return` statement on line 5. It causes the element to be inserted multiple times into the set.

Evaluation

Transition analysis  Although the erroneous transition is printed in the group `all(neg)`, there are almost all transitions of the model, so this is not very helpful. The more specific groups are either empty or unhelpful.

Invariant analysis  This type of analysis is not suitable for this problem, so we skipped it.

Transformation analysis  All three transformations suggest replacing the sequence of actions which adds a 1 (P: {1|1|0} is the incorrect state in our counterexample) with a sequence which starts to add a 2 and thus avoid the problem. Unfortunately, it does not lead any closer to the problematic transition.

Summary: useless

5.1.7. pushparent_2.dve

Error description:
The second error is exchanging lines 2 and 3, so an element already present in the set can be overlooked and inserted at the end of the list.

Evaluation

Transition analysis  This error is rather complex, it changes three transitions, so it’s not surprising that transition analysis does not output anything useful.

Transformation analysis  The transformations only suggest changing the initial state of the set so that adding the element does not result in an assertion violation.

Summary: useless
5. Algorithm Evaluation

5.1.8. rw_2.dve

Error description:
The error lies in the reader not releasing the writer lock when it is done, causing a deadlock on the next operation.

Evaluation

Transition analysis  At a first glance, the first interesting group (only(neg)) contains many transitions, but they’re just two copies of reader’s actions, one for the first reader and one for the second. It provides the same information as rw_1.dve.

Transformation analysis  As with rw_1.dve, the transformation analysis suggests removing reader’s action altogether.

Summary: touches

5.1.9. bakery_err1.dve

The bakery lock for 2 participants, coming from BEEM.

Error description:
The error is located in choose -> choose { guard j<2 and number[j]>max; effect max = number[j], j = j + 2;}, where we should increment just by 1.

Evaluation

Transition analysis  Here the transition analysis found the culprit and put it right into the cause(neg) group. Excellent.

Transformation analysis  All the transformations suggest changing the action where j is incremented by 2 to incrementing just by 1 by using different values in the number[] array.

The error is rather severe and the system in location choose can choose (deterministically) between two transitions, one correct and one incorrect, thus making transition analysis easier.

Summary: great
5. Algorithm Evaluation

5.1.10. airline_5_2.dve

This is the airline model from Software-artifact Infrastructure Repository recreated in the DVE language.

Evaluation

This is an example of a model which cannot be used with the explanation algorithm. Both the definition of negatives and positives require the last locations to be the same, but in this model, having the same locations means that we have reached the error. Therefore no positives can be found. The same problem occurs in models of mutual exclusion protocols where we are looking for the goal P1.CS && P2.CS. No positives can exist with the same locations as the counterexample, whose locations are P1.CS and P2.CS.

Summary: useless

5.1.11. nested_monitor_5.dve

This is the nested_monitor model from Software-artifact Infrastructure Repository recreated in the DVE language.

Error description:

The locking of buf_lock is superfluous and it causes the system to deadlock when a consumer tries to read before anything is written.

Evaluation

Transition analysis    There is just one negative, the original counterexample, which has only one action, so there is not much to compare.

Invariant analysis    There is no data in this model, except the synchronization primitives, so invariant analysis is unsuitable for this task.

Transformation analysis    All the transformations suggest inserting first some elements into the buffer, because then the consumer does not deadlock when trying to get a value. The only information this provides is that it’s the consumer who blocks.

Summary: useless
5. Algorithm Evaluation

5.1.12. array_partition_4.dve

This is the array_partition model from Software-artifact Infrastructure Repository recreated in the DVE language.

Error description:

One of the transitions, go_up -> go_up { guard lo <= hi && a[lo] <= pivot; effect lo = lo + 2 } is incorrect, it should add just 1 to lo. This way it skips some elements, specifically in this run it skips the 1 at position 1 ending up with an array that is not divided by the pivot (0) into two halves.

Evaluation

All the positives have such initial values of the array that the error does not occur.

Transition analysis  The only nonempty groups are all(neg) and all(pos), both contain, among others, the problematic transition. The other groups are empty which follows from what kinds of positives were found. No useful information here.

Invariant analysis  We placed an instrumentation point before the start of partitioning and another one at the end of the algorithm. There are many invariants and it’s hard to find any relevant to the error.

Transformation analysis  All the three smallest transformations do the same thing, change the initial data so that the erroneous program ends up in the correct state as written above.

Summary: useless

5.1.13. array_partition_4b.dve

Error description:

The error is located in the transition swap -> init2 { guard lo > hi; effect tmp = a[hi], a[hi] = a[lo], a[lo] = tmp; } where there is < instead of a >. This causes a deadlock in cases where lo < hi, because no transition is defined for that case.
5. Algorithm Evaluation

**Evaluation**

**Transition analysis**  The group `only(pos)` contains just two transitions, one of them being the incorrect one. This may imply that the positives managed to avoid the deadlock by taking this transition, and one might infer that the condition has a problem. But it's certainly not a direct line of thought.

**Invariant analysis**  We placed one instrumentation point before the algorithm starts. Like in the previous run, the positives differ from the negatives in selecting an initial configuration which does not cause the error to happen. Unfortunately, the invariants are unable to capture this or provide any insight.

**Transformation analysis**  All three transformations are practically identical and again, simply replace generating problematic input variables with inputs that don't cause the error to occur.

*Summary: touches*

5.1.14. **ConcurrentLinkedQueue**

This is a model of the `offer` method from the class `java.util.concurrent.ConcurrentLinkedQueue` from OpenJDK, obtained from http://hg.openjdk.java.net/jdk7/jdk7/jdk/raw-file/00cd9dc3c2b5/src/share/classes/java/util/concurrent/ConcurrentLinkedQueue.java

The method is faithfully translated into DVE, using some m4 macros to make the translation process faster.

**Error description:**

This version of the model has an error where we use a nonatomic compare and set instead of `p.casNext(null, n)`, causing a race condition. The model checker finds a counterexample where this race condition occurs and the system ends up in an incorrect memory state.

**Evaluation**

**Transition analysis**  This analysis outputs many transitions in the any groups and some transitions in `only(pos)`. From there we can see that the negatives never take the first or third branch in the innermost `if` statement (the compound one, with three branches and two `ifs`). So the negatives always take the middle
branch, because `p.next == null`, for all processes. This might be a hint at the race condition, but probably not very strong.

**Invariant analysis**   Not used.

**Transformation analysis**   The transformations suggest a different interleaving of parts of negatives and positives, correctly indicating the race condition.

*Summary: notbad*

**5.1.15. ConcurrentLinkedQueue_2.dve**

**Error description:**

There is an error when accessing memory, instead of looking at location `x`, we look at `x + 2` in the action `p = succ(p);`

**Evaluation**

**Transition analysis**   This analysis shows 6 transitions in `only(neg)`, 3 for the second and the same 3 transitions for the third process, so the user only has to analyze 3 transitions if they notice this symmetry. The three transitions represent one branch of the if statement, the one where the problem is located and indeed, one of them is the “guilty” transition. We can conclude that this analysis was successful.

**Invariant analysis**   Not done.

**Transformation analysis**   Corresponds to transition analysis – the transformations suggest replacing a trace containing the erroneous transition with one which does not contain it.

*Summary: notbad*

**5.1.16. ConcurrentLinkedQueue_3.dve**

**Error description:**

This version of the model also contains the iterator, with an error where `nextItem` is not set before returning an element.
5. Algorithm Evaluation

Evaluation

Transition analysis The transitions given in only(pos), only(neg) are not relevant to the error.

Invariant analysis The list of invariants is longer than the model itself, so it’s not helpful in any way. We can notice, however, that \( I_0.\text{count} \) is always smaller than 3 in the positives, which means that the assertion is true, because the left side of the implication is never true. Which in this case means, that the positive is still an incorrect run of the system.

Transformation analysis The three transformations are all identical, only occurring in different negatives. They change the negative so that the count is smaller and the implication in the assertion can be thus satisfied. It’s not useful for finding the error.

Summary: useless

5.1.17. rw_3.dve

Error description:

In this version, we don’t lock the \( r \) mutex atomically, which gives an opportunity for a race condition. This results in two processes being in the CS at the same time.

Evaluation

Transition analysis There is one transition in only(neg), unfortunately it has nothing to do with the error. The remaining groups, all(pos) and all(neg) both contain many similar transitions, so they are not interesting.

Transformation analysis The first two transformations use positives that also contain the error and they don’t contain the problematic place at all. The last one uses a good positive, but it does not show the race condition that occurs.

Summary: useless
5. Algorithm Evaluation

5.1.18. rw_4.dve

Error description:
The writer does not acquire the writer lock atomically, leaving an opportunity for a race condition.

Evaluation

Transition analysis Only(neg) contains two transitions, but they are not related to the error. All(neg) contains the two erroneous transitions, but they are not immediately noticeable among the total 5 transitions.

Transformation analysis The first two transformations contain a positive that uses the erroneous transitions to reach a state which does not violate the assertion. These cases cannot be used for finding the error. The last transformation correctly replaces a part of a negative containing the race condition with a simpler part of a positive.

Summary: touches

5.1.19. rw_5.dve

Error description:
Here two lines of are exchanged, readers = readers + 1 and if (readers==1) writer.lock()..

Evaluation

Transition analysis There is one transition in cause(neg), but it is not relevant to the problem. Only(neg) contains some more transitions, one of them is the exchanged action, but it is not obvious that there is the problem from the listing.

Transformation analysis Two transformations suggest completely removing a part of negative in which a reader enters and leaves the CS. The third suggests a different interleaving of two readers where the second reader performs the actions add → cnt → q1 (the ones with the error) later rather than sooner. This results in the writer being unlocked and the deadlock does not occur. This may be helpful information.

Summary: touches
5. Algorithm Evaluation

5.1.20. rw_6.dve

Error description:
The error in this model was designed specifically to be detected by transition analysis. In the writer process, there are two transitions that are almost equal, except for the error. The system can take either of them, making it possible to find a positive with the correct transition and a negative with the incorrect transition.

Evaluation

Transition analysis  The algorithm correctly identifies the erroneous transition and shows it in cause(neg).

Invariant analysis  We placed an instrumentation point at the end of the counterexample. In positives, there is an invariant which states that the local and global copies of X are equal. This invariant is not present in the negatives, which is correct. The remaining invariants are the same for negatives and positives.

Transformation analysis  All the transformations suggest removing the sequence of actions of the writer where it uses the incorrect transition.

Summary: great

5.1.21. rw_7.dve

Error description:
In this version, we have omitted the if (readers == 0) condition before releasing the writer lock. This causes a violation of the mutual exclusion property.

Evaluation

Transition analysis  The group only(pos) contains one transition, but it’s not the one with the problem. However, it works with the writer mutex, which may indicate that there’s something wrong with that, which is true.

Transformation analysis  All three transformations use a positive that also contains the error and don’t make much sense and are not useful in understanding the error.

Summary: touches
5. Algorithm Evaluation

5.1.22. train_gate_corrected.dve

A corrected version of train_gate model where we fixed the problem with the global variable e.

Error description:

There is a race condition caused by incorrect modelling of the system. When the gate is in state S6, where it got an appr signal and is about to send a stop signal, the train continues to move and crosses the bridge. The gate is then unable to send the stop signal. In the Uppaal model, the state S6 is committed, which does not allow the train to move until the gate leaves S6. DVE also has committed states, but the semantics is different and it cannot be used.

Evaluation

This run was examined without prior knowledge of the error.

Transition analysis  The analysis shows that in positives the train 5 arrives before train 1 whereas in negatives it happens in the opposite order.

Transformation analysis  The output of this analysis corresponds directly to that of transition analysis. In the end, we found the error just by analyzing the counterexample.

Summary: useless

5.1.23. termination.dve

An incorrect attempt at constructing a termination detection algorithm for a situation where two processes communicate using shared memory. The goal of the processes is to set all values in their portion of the memory to 0 while the processes walk through the memory sequentially. At any time, one of the processes can put a 1 in the other process’s memory. Termination is detected by remembering if this happened during the last swipe.

Error description:

When nothing is changed in the partner process’s memory during a round, we detect termination. However, this is not enough as we may still need to act upon the 1’s our process received during the current round. This error is rather hard to detect, we don’t expect explain to be very successful.
5. Algorithm Evaluation

Evaluation
The explain algorithm finds only two positives, so it does not have much information for analysis.

Transition analysis Only(neg) is the most interesting group in this output. It contains the same three transitions from each process. It's the transitions that something was sent and that termination has not happened. This fact may imply that nothing happens there. And that's actually true, both positives maintain the memory full of zeroes.

Transformation analysis All the three transformations suggest removing a part of the execution where a 1 is inserted in the memory. This corresponds to the nature of the positives found by the algorithm.
Summary: useless

5.1.24. nested_monitor_5a.dve

Error description:
Producer handles the queue incorrectly, decreasing both the empty and full counters.

Evaluation
Transition analysis Almost all groups except all(neg), all(pos) are empty which makes this analysis useless.

Transformation analysis The transformations suggest removing a part of execution where the variable e_empty reaches 0 which causes the deadlock. That sequence of actions is inevitable though, so transformation analysis isn't helpful either.
Summary: useless

5.1.25. nested_monitor_5b.dve

Error description:
In this version, the consumer does not use an atomic increment of the variable e_empty. This naturally leads to incorrect values and eventually to a deadlock.
5. Algorithm Evaluation

Evaluation

Transition analysis  Almost all groups except \textit{all(neg), all(pos)} are empty which makes this analysis useless.

Transformation analysis  Even though this is a race condition, all transformations just suggest removing a part of the execution, making the analysis unhelpful.

\textit{Summary: useless}

5.2. Summary

The complete statistics for the one-word summary is given in the table. In most cases explanation is unsuccessful.

\begin{center}
\begin{tabular}{llr}
\hline
evaluation & count & \% \\
\hline
useless & 12 & 48 \% \\
touches & 8 & 32 \% \\
notbad & 3 & 12 \% \\
great & 2 & 8 \% \\
\textit{total} & 25 & 100 \% \\
\hline
\end{tabular}
\end{center}

5.2.1. Successful runs

The five successful runs were

- \texttt{lamport\_47.244.dve}
- \texttt{bakery\_err1.dve}
- \texttt{ConcurrentLinkedQueue.dve}
- \texttt{ConcurrentLinkedQueue\_2.dve}
- \texttt{rw\_6.dve}

All of them except \texttt{ConcurrentLinkedQueue.dve} have found the error using transition analysis. That was possible because the models contained an alternative to the erroneous transition and therefore the error could be found by comparing transitions from positives with those from negatives. In \texttt{rw\_6.dve} this kind of error was inserted on purpose, in the other cases the errors just happened to be this way.

In \texttt{ConcurrentLinkedQueue.dve} the error is a race condition. In this case, transformation analysis was able to find it. Similarly in \texttt{rw\_4.dve}. In the remaining runs with a race condition, \texttt{explain} was not able to find the cause. In
5. Algorithm Evaluation

train_gate_corrected.dve the error lies in incorrect modelling of the system and the transformations given by the analysis do not point to the real error. The remaining runs containing a race simply didn’t show any useful transformation. Either the positives used were actually also wrong or were not similar enough to show the problem. Maybe a more specific condition for selecting the positives could help.

5.2.2. Error kinds in other models

Furthermore, we have three instances of the ConcurrentLinkedQueue model with different errors. In the first two cases the error is located in the code that adds an item, in the last case it’s in the iterator. The results are not bad for the first two even though they have completely different types of errors at different lines of code. The error was not found in the instance with the iterator.

There are three instances of the nested_monitor model, each of them contains a different error. All of them were evaluated as useless. What the instances nested_monitor_5a.dve and nested_monitor_5b.dve have in common is that transition analysis outputs transitions only in the groups all(neg) and all(pos). The reason is that the sets of transitions in positives and negatives are the same which is caused by the fact that a process always executes the transitions in a sequence.

The models pushparent_2.dve and rw_5.dve contain the same type of error – exchanged lines of code. Aside from this, the errors and code are not very similar. In rw_5 the error is not that severe, with certain values of the variables the deadlock may not occur, which makes it more likely for useful positives to be found. Unfortunately the analysis wasn’t very successful in pointing at the error. In pushparent_2 there are also valuations on which the error does not show up, but the runs with these valuations are uninteresting, because those values are assigned at the beginning before the “main” code is run as opposed to rw where the values are a result of running the “main” code.

The errors in models bakery_err1, array_partition4 and ConcurrentLinkedQueue_2 are also similar, adding a 2 instead of a 1 to a variable. Explain was successful on two out of the three models. We already described the reasons for successful explanation above, so let’s look why it failed in array_partition. The groups only(neg,pos) and cause(neg,pos) are empty, which means that both positives and negatives contain the same transitions. That is easily possible, because for some valuations the same sequence of actions as in the negative can lead to a correct result.
5. Algorithm Evaluation

5.2.3. Error kinds in the readers/writers lock

We inserted six different errors in the model of the readers/writers lock. \texttt{rw\_1} and \texttt{rw\_2} contain the same error, but \texttt{rw\_1} has only one reader, which reduces the number of transitions in transition analysis, because the reader processes are symmetric and no detection of multiple instances of the same transition is performed.

\texttt{rw\_3} and \texttt{rw\_4} have a similar error – not locking a mutex atomically. In the first case, nothing useful is found, in the latter one transformation demonstrates a race condition. This distinction exists even though the errors are in principle identical, only located in different places of the model.

The errors in the remaining instances of \texttt{rw} were not similar to each other. Once it was swapping two lines of code and in the other case removing an if guard. Analyses of both provided only some very indirect clues.

As a summary, the type of error is as important as the model itself. In some cases, we can get very good results but usually the results are rather poor.

5.2.4. Common properties

There are a few common traits of the models and the outputs explain generates on them. Some of them cause explain to fail, others may not have any influence on the outcome. We will summarize the common properties in this section.

**Too few positives or negatives** This is likely a problem, as we may have insufficient data to perform the analyses accurately. Occurs in \texttt{nested\_monitor\_5}, \texttt{train\_gate\_corrected} and \texttt{termination}.

**Failing positives** Positives may pass through the error location and still be useful for diagnostics, but in some cases it does not work very well. This is the case of \texttt{rw\_4}.

**Positive misses the problem** Positives may also completely circumvent the problem and thus be useless for any comparison with negatives. This is a very subtle distinction and it depends highly on the model and the kind of error it has. Occurs in \texttt{pushparent\_1}, \texttt{pushparent\_2}, \texttt{Concurrent\_LinkedQueue\_3}, \texttt{rw\_3}, \texttt{rw\_7}, \texttt{termination}, \texttt{nested\_monitor\_5b}.

**Too many invariants** The tool outputs so many invariants that even though some of them may be insightful, it takes a lot of work for the user to consider them all. This happens practically in all models where invariant analysis was used.
5. Algorithm Evaluation

Too many transitions Transition analysis outputs too many transitions. This is similar to the previous problem except that it does not happen so often. Occurs only in lamport_47.244.

Transition and transformation analysis say the same thing This is not a problem per se, it just means that transformation analysis brings no new information. Occurs in train-gate_53.277, rw_1, ConcurrentLinkedQueue_2, train_gate_corrected.

Symmetric transition analysis Transition analysis prints very similar transitions, usually belonging to multiple instances of the same process. If the algorithm had information about multiple instantiations of a process template, this could be handled automatically. Occurs in rw_2, termination, ConcurrentLinkedQueue_2.

Symmetric transformation analysis The state spaces are usually symmetric to a degree and so are paths in the graphs. Unfortunately this means that additional transformations bring no new information. Occurs in lamport_47.244, array_partition_4b, ConcurrentLinkedQueue_3.

Empty transition analysis The sets only(neg/pos), cause(neg/pos) contain none or too few transitions which naturally means that the analysis is useless. It happens when the sets of transitions in positives are the same or very similar to those in negatives. Occurs in pushparent_2, array_partition_4, rw_3, rw_4, rw_7, nested_monitor_5a, nested_monitor_5b.

5.2.5. Use of the Extension algorithm

In the presented tests, we have used a search depth of 100 which was too large for the Extension algorithm to be used. There were some models with a large enough state space, but as the matching of the state with the counterexample always begins at this depth and counterexamples were too short, the algorithm was not run there either. We tried decreasing the search depth for the larger models and the results are available in the database on the enclosed CD. In both cases though, there were no improvements in the quality of explanation. There were fewer positives and negatives which resulted in fewer transitions listed in transition analysis. The transformations were shorter, but provided the same information as in the deeper search.
5. Algorithm Evaluation

5.3. Performance

We measured the run time of the program on a multicore machine using three different models and using a different number of CPU cores. The table 5.1 contains all the results. It contains run times of parts of the entire algorithm in seconds. The parts not included in the results such as Extension, Transformation and Transition analyses were skipped because their run time was negligible. See the complete run results on the CD for complete information. The lamport model ran out of memory when run in a two-core configuration, possibly due to a different order of exploration.

The time in the column explain is the time taken to find negatives and positives by the basic algorithm. The time in the column inv. a. is the length of invariant analysis, dominated by invoking Daikon.

We used a machine with a quadcore Intel Xeon 5130 CPU on 2.00GHz and with 16 GB of RAM to perform the measurements. Search depth was set to 100 and up to 50 positives and 50 negatives were found.

![Figure 5.1.: Run times](image)

The table 5.2 gives an idea of the sizes of the models. We can see that simple reachability is noticeably faster than the search for positives and negatives.

![Figure 5.2.: Model sizes](image)

Specific arrangements were necessary to obtain repeatable results. It was not sufficient to simply run the algorithm on the same input, because when multiple cores were used, states are processed in an nondeterministic order, which is likely to cause the program to find a different counterexample and
5. Algorithm Evaluation

make the explanation completely different. To mitigate this problem, we stored
the counterexample found in the singlecore case and used it for subsequent
parallel runs. For this reason, reachability times for multicore runs have no
informative value and we only include the time for Explain. This approach
required a change to the program code which can be found in a patch called
multicore_provisions on the CD.

Figure 5.3.: Graph of run times
Bibliography


Bibliography


A. Contents of the CD

- **mainstore** Contains complete records in Python source format of all the performed tests along with complete inputs and outputs.

- **scripts** Python scripts for reading the reports.

- **explain** A Darcs repository containing complete sources of DiVinE 2 including the implementation of the counterexample explanation algorithm.

- **text** This thesis in source and PDF forms.
B. Reading test reports

The reports are stored as nested Python dictionaries in Python source files. The data is split into multiple files to avoid dealing with a single large file.

B.1. Reading the data in clean Python

To read the data without using any additional scripts, the following Python code is sufficient. Any recent version of Python should work.

```python
import datetime
data = eval(open('data<x>.txt').read())
```

where `<x>` is substituted with the number of the data file to read. The variable `data` then contains a list of dictionaries which can be further processed.

B.2. Using the GUI

The `scripts` directory on the CD contains a simple GUI for working with the data. To use it, Python 2.6 and wxPython is required. The GUI can be started directly from the CD by setting `scripts` as the working directory and running the Python script `gui.py`:

```bash
$ cd scripts
$ python gui.py
```

Setting the working directory correctly is important, the script expects the data to be found in `../mainstore`. 