Institut für Informatik
der Technischen Universität München

Methods for the Diagnosis and Automatic Repair of Software Systems

Christian Kern

TECHNISCHE UNIVERSITÄT MÜNCHEN
Chair for Foundations of Software Reliability and Theoretical Computer Science

Methods for the Diagnosis and Automatic Repair of Software Systems

Christian Kern

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Helmut Seidl

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Javier Esparza
2. Prof. Keijo Heljanko, Aalto Universität / Finnland

Die Dissertation wurde am 22.02.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29.05.2013 angenommen.

Abstract

Software bugs account for the majority of costs in the software development process. In this thesis we make two contributions to the general problem of automatically diagnosing and correcting bugs. In the first contribution, we present a lightweight approach for repairing standard bugs in program code. The programmer defines a catalogue of syntactic constructs she consideres error prone, together with suitable alternatives. Given a faulty program, search techniques are applied to repair the program.

The second contribution are model-based approaches for the diagnosis of failures in partially observable, distributed systems that trigger alarms. We are interested in the possible behaviours that lead to an alarm as basis for deriving an explanation for the bug. A natural formal model for concurrent systems are Petri nets. An "unrolled" representation of such a net, its unfolding, encapsulates all its behaviours in a compact way. A prefix of the unfolding is used to represent the behaviours that lead to an alarm. Since the computation of this prefix is algorithmically involved, we give a method for computing an overapproximation that trades precision for speed. We report on an implementation that constructs this prefix when the observation sequence is known, but also present an online and forward-looking approach and compare these approaches. Moreover, we present SAT solving methods for producing an explanation. Finally, we focus on the Reveals relation (introduced by Haar), that establishes an inevitability relation between system events in an unfolding. This supports reasoning about the occurrences of non-observable events purely by using the observation. We notably improve the existing decidability result for Reveals by introducing a tractable algorithm for deciding it. Moreover, we present a highly efficient implementation of this algorithm.

Zusammenfassung

Softwarefehler sind für den Großteil der Kosten im Softwareentwicklungsprozess verantwortlich. In dieser Arbeit stellen wir zwei Ansätze für das Problem der automatischen Diagnose und Korrektur von Softwarefehlern vor. Der erste Beitrag ist ein leichtgewichtiger Ansatz; ProgrammiererInnen definieren einen Katalog von syntaktischen Konstrukten, die sie für fehleranfällig halten, zusammen mit möglichen Alternativen. Um das defekte Programm zu reparieren werden Suchmethoden verwendet.

Der zweite Beitrag sind modellbasierte Ansätze für die Diagnose von Fehlern in partiell beobachtbaren, verteilten Systemen, die Alarme auslösen. Als Basis für die Fehlererklärung sind wir an den Systemverhalten interessiert die einen Alarm auslösen. Ein natürliches Modell für nebenläufige Systeme sind Petrinetze. Die "entrollte" Darstellung eines Netzes, seine Entfaltung, fasst Verhalten des Netzes kompakt zusammen. Ein Entfaltungsprefix wird benutzt um alarmauslösende Verhalten des Netzes darzustellen. Dessen Berechnung ist algorithmisch sehr komplex. Daher wird eine Überapproximation, bei der Genauigkeit gegen Geschwindigkeit getauscht wird, berechnet. Wir berechnen diesen Prefix, zum einen wenn die Beobachtungssequenz eines Alarms schon feststeht, zum anderen vorausschauend während das System läuft. SAT-Solving Methoden ermöglichen es, eine korrekte Fehlererklärung von diesem Prefix abzuleiten. Im letzten Teil der Arbeit beschäftigen wir uns mit der Reveals-Relation (siehe Haar), die eine Unausweichlichkeits-Relation zwischen Ereignissen in einer Entfaltung definiert. Diese Relation hilft über unsichtbare Ereignisse zu sprechen, nur unter Benutzung der sichtbaren. Wir verbessern die Entscheidbarkeits-Ergebnisse für die Berechnung von Reveals merklich, indem wir einen in der Praxis berechenbaren Algorithmus vorstellen, zusammen mit einer äußerst effizientem Implementation.

Acknowledgements

First of all, I would like to thank my supervisor Javier Esparza for giving me the opportunity to become a student at his chair. His door was always open for me. He was taking much time for guiding me the right way and answering any questions. Without his continuous and persistent support, this thesis would not have been possible. His support in any way was better than I could have ever asked for. He did a great job and I learned very much from him, not only with respect to research.

During my research visit to Cachan in France, Stefan Schwoon was my host. His hospitality was very kind and I felt very welcome at his research group. He showed me new directions for my research and was a very important part for it. Thank you very much for that! In this context, I also would like to thank Stefan Haar for giving me the great opportunity of doing research with him. Moreover, thank you Keijo Heljanko for reviewing my thesis. Further thanks goes to all my friends and colleagues at the chair of Prof. Esparza for providing such a pleasant working environment. Thank you Andreas Gaiser, Stefan Kugele, Jan Křetínský, Michael Luttenberger, René Neumann and Maximilian Schlund, it was an honour to work with you.

My research was supported by a scholarship of the PUMA graduate school. Thank you, Helmut Seidl, for letting me be a member of this excellent graduate school. The discussions with other members of PUMA as well as the variety of offered lectures were very inspiring and an important part of my research.

A very big thanks goes to my parents and my sister for their support and love, not only during the time working on this thesis, but my whole life. Last but not least, I want to thank you, Martina, for your everlasting support, your trust in me and your kindness during all those years.

Contents

Chapter 1

Introduction

This thesis deals with the topic whether and to what extent the task of debugging can be automated. Debugging refers to the task of fixing bugs. The term "bug" goes back to the year 1947, where Grace Hopper was working on the Mark II, a relay computer. This computer was not working as specified but the reason for this error was not some programming or wiring error. It was a moth trapped between two relays. Hopper wrote to her log [27]: "First actual bug being found". Nowadays, computer systems are very reliable. It is very unlikely that some insect is the cause for a software error. On the other hand, software systems have become very complex. Software failures are mainly caused by the programmer making mistakes. Still, by abuse of the term bug, such a programming error is called a *bug* and the task of detecting, finding and fixing such an error is called *debugging*. Debugging is a very time intensive and tedious task and so its automation is a valuable research topic. In this chapter, we first give an overview over our approaches and concretely discuss our contributions.

Figure 1.1 shows the schematic overview of our general approach. We assume a *software system* given as source code together with its complete *specification* (most left in the figure). The system and the specification are compiled in such a way that both are executable. We assume the system is *partially observable*, i.e. we are only able to observe the systems behaviour up to a certain degree. This additionally implies that it may be possible that the specification cannot be completely verified.

There, *diagnosis* comes into play. Diagnosis subsumes *monitoring*, the task of observing and recording the systems behaviour. The system emits

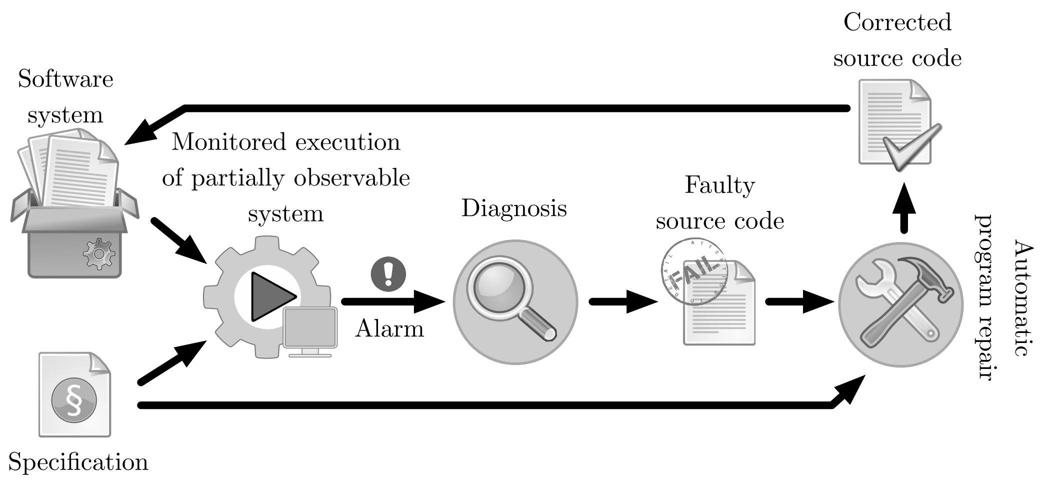


Figure 1.1: General goal

alarms, i.e. potential violations of the specification. Diagnosis is the task of analysing the recorded system behaviour. Spurious alarms, i.e. alarms that do not refer to violations of the specification, are sorted out and real faults that already happened or that are in the systems futures are analysed to find and report their cause. If the fault is projected to the systems future, it may be possible to avoid it. Otherwise, we have a faulty piece of code that needs to be repaired. We then deal with the question of how to automate this task, i.e. given a faulty program together with its specification, how one can automatically derive a correct program. This is called *automatic program repair*. The output of this procedure is the corrected source code. The software system can be updated with the repaired source code and the whole procedure is iterated.

This thesis contains *two specific contributions* to this general problem, one for program repair and one for diagnosis. The contribution for automatic program repair deals with the problem directly on the code level. We are interested in a practical approach and implementation details. This is therefore a very technical contribution. Contrary to that, the contribution to diagnosis concerns models of software, in particular Petri net models. As well as in the contribution for program repair, we also introduce and discuss algorithms together with their implementations but we additionally give a series of theoretical results. Clearly, the model based approach is therefore the more substantial part of this thesis.

For the contribution to automatic program repair, a testing based approach using an explicit model is presented. We are given a defective program as Java source code and we simulate different program variants until we find a correct one (with respect to the given complete program specification). We base our implementation on the explicit state model checker JPF[1] to achieve this. An explicit model checker does not use a sophisticated model for the software system, the states in the model are represented by a core dump of the virtual machine that is used to execute the program. This approach is discussed in detail in Chapter 2.

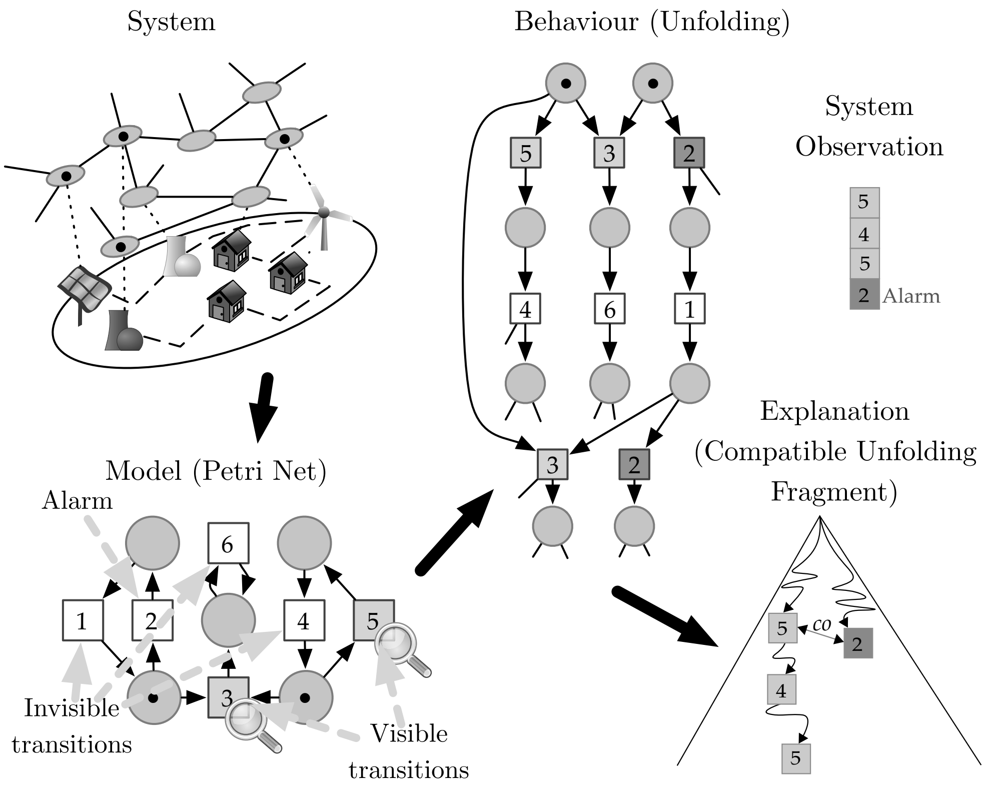


Figure 1.2: Approach - diagnosis with Petri nets

For diagnosis, as already mentioned, instead of considering code directly, we consider model-based approaches. Petri nets are natural models for distributed systems. Figure 1.2 shows our approach. We are given a system

[1]http://javapathfinder.sourceforge.net

that is modelled as Petri net and is only partial observable. The partial observability is modelled by partitioning transitions of the net into visible and invisible transitions. Invisible transitions are not observable whereas visible transitions are observable and can additionally be alarms. For analysing the possible behaviours of the system it is necessary to explore and store its state space. Therefore, as usual when doing model checking, we have to combat the state explosion problem. In distributed systems, different orders of transitions can lead to the same state. So, for fighting the state explosion problem, only the partial order of transitions needs to be stored. This is called partial order compression and the *unfolding* of a Petri net is a well-known representation for such a partial order compressed view on the behaviour of the net. When given the observation that leads to an alarm we want to analyse the alarm, therefore the behaviours of the net that show the alarms observation have to be computed and analysed. The compatible unfolding, i.e. the unfolding that represents the behaviours of the net showing the observation, is utilized for this purpose.

We proceed as follows. In Chapter 3 we present common preliminaries for our approaches on this topic and give related work on Petri nets and their unfoldings. In Chapter 4 we consider the presented general approach. A parallel Sokoban game is used as system for a case study. The system is modelled as product of transition systems, represented as Petri net. Its unfolding is used as behaviour representation. We introduce a heuristically approach for doing diagnosis while the system is running, i.e. while observing the system, the system's unfolding is synchronized with it. Additionally, an approach is presented not only considering the systems history but also its possible future behaviour. This enables speculating about failures in the systems future, but also adds the problem of removing behaviours that are invalidated when the system progresses. We present a solution for this problem; the respective approach is called *proactive diagnosis*. The subsequent Chapter 5 deals with the reveals relation, a relation supporting doing diagnosis using unfoldings. This non-trivial relation is, loosely speaking, formulated as follows: "a reveals b iff, whenever a occurs, b will eventually occur or has already occurred". Having this relation at our disposal, from the occurrence a we can derive the potentially invisible occurrence of b. The relation itself was introduced by Stefan Haar. Decidability results as well as the efficient computation of this relation are subject of this chapter.

Former versions of the contributions of this thesis were published in:

- Christian Kern, Javier Esparza: *Automatic Error Correction of Java Programs.* FMICS 2010: 67-81 [32].
- Stefan Haar, Christian Kern, Stefan Schwoon: *Computing the Reveals Relation in Occurrence Nets.* GandALF 2011: 31-44 [25].
  An extended version was accepted for publication as journal article in *Theoretical Computer Science.*
- Javier Esparza, Christian Kern: *Reactive and Proactive Diagnosis of Distributed Systems Using Net Unfoldings.* ACSD 2012: 154-163 [14].

In Papers [32] and [14], all case studies and most of the theoretical results were done by myself, as well as the implementations and algorithm design for Paper [25]. The major part of proving the improved bound in [25] was done by Stefan Schwoon and Stefan Haar. In Section 5.3 of this thesis, we present a rewritten and much more detailed version of this proof.

Chapter 2

Automatic Program Repair

Contrary to the common opinion that most of the time (and therefore money) in the software development cycle is spent on implementing new features, most of the time (50% or more) is spent purely on debugging, i.e. detecting and fixing bugs, as for example shown in [26] or in the extensive survey [34]. In this chapter we deal with this question and present our heuristically approach towards solving it. Before presenting our approach, we give an overview over the wide-ranging research fields of bug localization and program synthesis. Our approach is located between both areas.

Bug localization (sometimes called bug interpretation) and bug fixing have been intensively studied in the last years. For this research field we follow the notion presented in [54]. A *bug* is a defect in the program code caused by an infection. An *infection* is a state in some execution of the program that differs from the state as it was intended by the developer. Such an infection may not lead to undesired program behaviour, the program can still fulfil its specification if, e.g. the infection does not propagate. However, if it becomes visible to an observer (e.g. the output is wrong), we call this a *failure*. The task of bug localization is therefore the task of locating the code location that is responsible for the infection.

A program *trace* is the sequence of program states in a program execution, leading from an initial state to a target state. Traces reaching an error state are called error traces, all other traces are called successful. Several proposals for bug localization are based on the idea of capturing differences between error traces and successful traces of a program. Zeller [55] introduces a technique called Delta Debugging, eliminating possible failure causes by automatically and iteratively testing hypothesis on the failure. This general technique is used by Cleve and Zeller[7] to compare the intermediate program states of error and successful traces and apply Delta Debugging to find a minimal set of value changes in variables, transforming a successful run into a failing run. This information is used to report possible infections to the user.

Ball et al. [1] search for transitions occurring in multiple error traces but no successful trace, which are suspected to be infections. Their approach is implemented in the SLAM Toolkit [2]. Similar techniques [23] have also found its way into the Java Pathfinder [50] model checker.

Groce et al. [22] propose a notion of distance between traces. SAT is used as program model. Distance between two traces is defined as the number of different assigned SAT variables that are representing program variables. Pseudo boolean constraints are added to the SAT formula for generating a closest pair containing a successful and an error trace using a pseudo boolean solver. The infection(s) is/are assumed to be localized at the differences between these traces. This approach is implemented using the CBMC [6] model checker.

Tarantula [30] visualizes differences between successful and error traces: program statements are colored accordingly to the ratio between how often they are visited by successful traces, and how often by failure traces. If the ratio "visited in a error trace" and "visited in a successful" trace is very high, this might be an infection.

Further proposals for bug localization only use information from error traces. Wang et al. [51] determine a causality-chain inducing the error by applying an algorithm for computing pre-conditions to an error trace. Griesmayer et al. [21] consider systems with several components and propose an iterative procedure that considers one error trace at a time and uses it to narrow down the set of components that can be responsible for the fault.

A common advantage of all these approaches with respect to our proposal is the absence of assumptions about the cause of the bug, compared to our assumption that bugs are located at hotspots. However, the absence of assumptions also makes automatic repair problematic, and in fact none of the approaches above explicitly studies it.

The two approaches closest to our work present proposals for automatically localizing *and* fixing bugs. Weimer et. al [52] assume that the bug can be fixed by deleting, inserting or swapping instructions in the source code.

Genetic algorithms are used to generate program variants, which are then sequentially tested for being correct. Instead of applying genetic algorithms, we generate one single meta-program embedding all variants, and explore it using search techniques. We suspect that this approach is more adequate for bugs requiring changing the code at several places; however, a detailed comparison is problematic, because genetic algorithms can be tuned according to a wide range of parameters, and is beyond the scope of this thesis. Jobstman et al. [29, 48] reduce program repair for finding a winning strategy in a game, and present impressive benchmarks, albeit mostly in the hardware area. Our approach can be seen as a special case of their technique that can be implemented on top of JPF with reasonable effort, allowing to profit from all the algorithmic expertise embodied in it.

2.1 Introduction

Our approach follows from the observation of the debugging behaviours of software engineers. When debugging, they often focus on "hotspots" in the program code, i.e. code locations where bugs are likely to occur. Then they check if a change of the code at these "hotspots" may correct the bug.

A typical example of hotspots are comparisons of integer expressions, which are likely to lead to "off-by-one" errors, like typing `x < 0` instead of `x <= 0`, or `for (int i = 0; i < N; i++)` instead of `for (int i = 0; i < N+1; i++)`.

We propose to automatize this approach. Instead of manually searching for hotspots, programmers just define a catalogue of syntactic constructs, like for instance `EXPRESSION1 < EXPRESSION2`, and for each of them a set of possible alternatives, like, for instance, `EXPRESSION1 <= EXPRESSION2`.

A *choice* at a hotspot is the decision for a specific alternative at this code location, including the unmodified source code at this hotspot. A *program variant* is derived by making a choice at each hotspot.

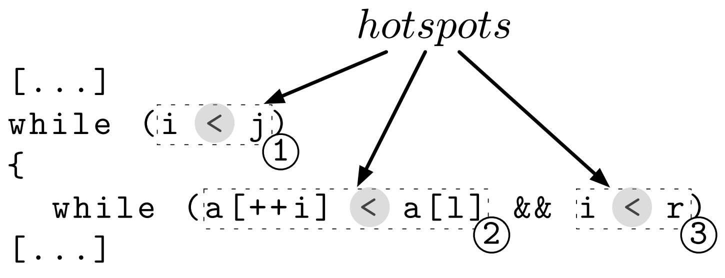
Furthermore, developers specify a set of test inputs, for instance by fixing the range of input variables. Some tool can then in principle generate program variants of the program generated by the alternatives, and test each of them on the test inputs, until some variant passes all the tests.

A naive way of testing all variants on all test inputs is to sequentially compile each variant and test it on all inputs. However, this approach is highly inefficient. We propose an approach, where we first generate a meta-program P_m that can simulate all the variants of P and then run P_m in a certain way on the set of test inputs, excluding variants along the way. We then present an implementation of this approach on top of the Java Pathfinder (JPF) model checker. In fact, the state exploration algorithm of JPF turns out to exactly meet the needs of our technique.

2.2 Selecting Hotspots and Alternatives

We use *syntactic code analysis* to search for hotspots; code locations where (part of) a bug could have been injected. As our implementation is based on the Java Compiler API[1], hostspots are subtrees of the parse tree of the program.

We explain our approach by means of an example. Figure 2.1 shows part of a sorting algorithm in Java. Assume that the hotspot heuristic in this example extracts binary expressions combined with the less-then comparator; i.e. all expressions of the form `EXPRESSION1 < EXPRESSION2`. For these two lines of code, three expressions of this form are found, namely hotspots ①-③.



```
[...]
while (i < j)
{
  while (a[++i] < a[l] && i < r)
[...]
```

Figure 2.1: Heuristically selected hotspots

For each hotspot, a *changeset entry* collects a set of possible alternatives, plus the original code. A *changeset* collects all changeset entries for a program.

For instance for the hotspots in Figure 2.1 we consider a heuristic that suggests `EXPRESSION1 > EXPRESSION2` as alternative to

[1]`http://java.sun.com/javase/6/docs/jdk/api/javac/tree/index.html`

`EXPRESSION1 < EXPRESSION2`. The result is the changeset displayed in Figure 2.2, i.e. a set containing a changeset entry for each hotspot.

CS = { ① → { `i < j`, `i > j` }, ② → { `a[++i] < a[l]`, `a[++i] > a[l]` }
③ → { `i < r`, `i > r` }}

Figure 2.2: Example changeset

We extract a *template program*, i.e. the original program where code of each hotspot is replaced with the template. See Figure 2.3 for that. A *program variant* is the result of replacing each template by one of the elements in its corresponding changeset entry.

2.3 Conflicting Hotspots

Two hotspots are *conflicting* if one of them is a subtree of the other. For instance, in Figure 2.4, hotspots ① and ③ are conflicting but ① and ④ are not. When heuristics produce conflicting hotspots we use Algorithm 1 to generate a changeset entry for the outermost hotspot, and use it for producing program variants.

Our algorithm constructs a forest F with hotspots as nodes. If ⓑ is a subtree of ⓐ in the program's parse tree we add to F the edge (ⓐ, ⓑ) and remove all transitive edges from the graph. We then generate a new changeset entry for each root of the forest by means of an iterative procedure. Given an edge (v_1, v_2) and elements e_1, e_2 of the changeset entries of v_1 and

```
[...]
while (①)
{
  while (② && ③);
[...]
```

Figure 2.3: Template program

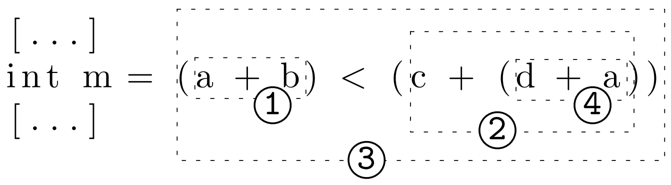


Figure 2.4: Conflicting heuristics

$$\mathrm{CS}_c = \{\ \textcircled{1} \rightarrow \{\texttt{a + b}, \texttt{a - b}\}, \textcircled{2} \rightarrow \{\texttt{c + (d + a)}, \texttt{c - (d + a)}\},$$
$$\textcircled{3} \rightarrow \{\texttt{(a + b) < (c + (d + a))}, \texttt{(a + b) > (c + (d + a))}\},$$
$$\textcircled{4} \rightarrow \{\texttt{d + a}, \texttt{d - a}\}\}$$

$$\mathrm{CS}_1 = \{\ \textcircled{1} \rightarrow \{\ \texttt{a + b}, \texttt{a - b}\ \},$$
$$\textcircled{2} \rightarrow \{\ \texttt{c + (d + a)}, \texttt{c - (d + a)}, \texttt{c + (d - a)}, \texttt{c - (d - a)}\ \},$$
$$\textcircled{3} \rightarrow \{\ \texttt{(a + b) < (c + (d + a))}, \texttt{(a + b) > (c + (d + a))}\ \}\}$$

Figure 2.5: Top: conflicting hotspots, Bottom: first resolution step

v_2, we denote by $e_1[e_2]$ the result of substituting e_2 for the code of v_2 in e_1. The procedure picks an edge (v_1, v_2) such that v_2 is a leaf, and replaces v_1's changeset entry by $Merge(v_1, v_2) := \{e_1[e_2] \mid e_1 \in Che_1, e_2 \in Che_2\}$, where Che_1, Che_2 are the changeset entries of v_1 and v_2, respectively; then the procedure removes v_2 from the graph, removes the changeset entry of v_2 from the changeset, and iterates.

Consider for example the source code in Listing 2.4 with the changeset CS_c of Figure 2.5/Top. The algorithm constructs the conflict graph $G = (V, E)$ with the set of edges $E = \{(\textcircled{3},\textcircled{1}), (\textcircled{3},\textcircled{2}), (\textcircled{2},\textcircled{4})\}$ and the set of vertices $V = [\textcircled{1}–\textcircled{4}]$. Changeset entry $\textcircled{4}$ is a leaf, and so its changeset entry is merged with the changeset entry of $\textcircled{2}$. The result is shown in Figure 2.5/Bottom. After repeating this procedure as long as possible, we get a merged changeset entry for program location $\textcircled{3}$.

2.4 Search Strategy

We have described an heuristically approach for identifying "error prone" program locations (hotspots) in a faulty program and suggesting alternatives.

Algorithm 1: ConflictResolution

```
input  : Set of conflicting hotspots $H_c$
output: Resolved set of changeset entries $R$
begin
    $V \longleftarrow H_c$;
    $E \longleftarrow \varnothing$;
    Directed Graph $G = (V, E)$;
    for $v1 \in V$ do
        for $v2 \in V$ do
            if $ProgramLocation(v1) \subset ProgramLocation(v2)$ then
                $E \longleftarrow E \cup (v1, v2)$
    $RemoveAllTransitiveEdges(G)$;
    while $\exists\ (v1, v2) \in E$ with $v2$ has no outgoing edges do
        $v1 \longleftarrow Merge(v1, v2)$;
        $E \longleftarrow E \setminus \{(v1, v2)\}$;
        $V \longleftarrow V \setminus v2$;
    return changeset entries of remaining vertices $V$
```

These heuristics return a changeset containing a changeset entry (a set of alternatives) for each hotspot. Now we show how to use the changeset to derive a corrected program. We generate test inputs, search the space of program variants obtained by a combination of changeset code replacements and select those variants satisfying the specification for all test inputs. The correctness of this set of variants can then be verified using some model checker. Since our implementation is based on the Java Pathfinder (JPF) model checker, we first discuss its search strategy.

2.4.1 Java Pathfinder

The JPF model checker is an explicit state model checker. Conceptually, JPF is a virtual machine that can simulate all possible runs of a program. Its input is a program P given as Java bytecode. Various techniques, e.g. state compression and partial order reduction are applied to fight the state space explosion. The state space of P is exhaustively explored using various search techniques. In this work, we focus on JPF's depth-first search implementation. We first discuss this approach, presented in Algorithms 2 and 3.

Algorithm 2: Java Pathfinder

```
input  : Program $P$
output: Is $P$ correct?
begin
    $s$ ⟵ new choice_point stack;
    while true do
        /* Program is executed until non-det. choice is
           possible or an end state is reached             */
        ;
        executeProgram($P$);
        if endState then
            if endState is errorState then return false;
            else if !Backtrack(s) then return true;
        else        /* Non-deterministic point in execution */
            choice_point ⟵ new choice_point;
            choice_point.doNextChoice();
            $s$.push(choice_point);
```

Usually, model checkers are used to analyze programs on all possible program inputs. This is achieved by introducing non-deterministic variables that symbolically represent value ranges. The program is verified on possible combinations of these values. For this analysis not having to explicitly execute the program with each value combination, various symbolic techniques have been studied and introduced.

In JPF, non-determinism is introduced either indirectly, e.g. when selecting the next thread that executes an action or directly, as statements in the source code under test. The program is executed until a non-deterministic choice is possible or the execution terminates. A stack that records made choices is maintained. For each non-deterministic choice in the execution, a `choice_point` is created on top of this `choice_point` stack, storing the different possibilities to continue the execution – those that have already been explored, and the current program state.

The execution is continued using depth-first search, i.e. the first choice not marked as explored is executed and marked as explored. If an error state is reached, a program failure together with an error trace is returned. If an end state is reached without errors, the search backtracks to the first `choice_point` to a choice point stack that has at least one unexplored choice. The program state of this `choice_point` is restored, and the execution continues with the unexplored choice. If no backtracking is possible, the program is declared correct.

Algorithm 3: Backtrack

```
input  : A reference to choice_point stack $s$.
output: Was backtracking possible?
begin
    while true do
        if $s$.empty() then
            return false;
        choice_point ⟵ $s$.pop();
        if choice_point has more choices then
            restore_state(choice_point);
            choice_point.doNextChoice();
            $s$.push(choice_point);
            return true;
```

2.4.2 Searchable Meta Program

In this section, we construct the already introduced meta-program P_m. Given P_m as input, JPF explores each program variant of program P for a given changeset on each test input. Note that all the steps described next for modifying the original source code are carried out automatically.

Test inputs are introduced utilizing non-deterministic choices in the source code under test. Consider for instance the example code in Figure 2.6 that calls the method `Verify.getInt(a,b)`. In context of JPF, this statement is not a Java expression returning a value, instead it represents a non-deterministic integer value within range $[a, b]$, resolved as stated in Section 2.4.1.

```
int[] a = new int[Verify.getInt(1,5)];
for(int i=0; i!=a.length; ++i)
a[i] = Verify.getInt(0,50);
```

Figure 2.6: Test input generator

Consider again the example code in Figure 2.6. Executed in JPF, all possible arrays `a[]` of size 1-5 and value entries 0-50 are explored. This piece of code can, e.g. be used as test input generator for sorting algorithms.

We explore program variants in a similar way. Recall that a changeset is a set of changeset entries, each of them consisting of a set of alternatives for a single program code location. We force JPF to explore each alternative in its depth-first search procedure. For that, we introduce a new statement: `Explore.getChoice(size, ident)`. In the context of JPF, this statement represents a non-deterministic value within the integer range [0, `size`−1], however, a non-deterministic value with the same identifier might get hit more than once within the JPF execution (due to loops and recursive calls), but the decision must be only made once. So, during the execution of the program under test, we only create a `choice_point` when no `choice_point` with the identifier `ident` exists in the choice point stack, i.e. only the first time a non-deterministic value of this kind is hit in a path of the execution with this identifier. In all subsequent hits, the current choice of the already existing `choice_point` is just looked up and returned.

For each changeset entry we introduce an `Explore` method call with `ident`, a unique identifier and parameter `size`, the number of alternatives contained in the respective changeset entry.

The value returned by `Explore` determines the program alternative of the changeset entry to be executed next. This is the motivation for only allowing one `choice_point` for each identifier, created in one run of the program. Otherwise, e.g. because of recursive calls, all variants of a changeset entry could be exhaustively explored again. We use the "`?:`" operator[2] to execute the specific program variant. Its simplified syntax is:

```
CONDITION ? EXPRESSION : EXPRESSION
```

[2]`http://java.sun.com/docs/books/jls/third\_edition/html/expressions.html\#15.25`

```
[...]
while ( Explore.getChoice(2, id1) == 1 ?
          i < j : i > j )
{
  while ( Explore.getChoice(2, id2) == 1 ?
            a[++i] < a[l] : a[++i] > a[l] &&
          Explore.getChoice(2, id3) == 1 ?
            i < r : i > r )
[...]
```

Figure 2.7: Searchable meta program

If `CONDITION` evaluates to true (resp. false), the first (resp. second) expression is evaluated and its result is returned. This conditional operator is applied recursively. Consider for example the expression

```
j==1 ? EXPR_1 : j==2 ? EXPR_2 : j==3 ? EXPR_3 : EXPR_0.
```

If variable `j` has value `n` with `n` $\in [0-3]$, then `EXPR_<n>` gets evaluated. For every changeset entry, we replace the expression at the respective program location by such a conditional expression. We use the return value of the `Explore` function as choice for a code alternative of each program variant. Figure 2.7 shows the code segment we obtained for the example in Figure 2.1 and the changeset in Figure 2.2. Observe that in this approach, alternatives and hotspots must necessarily be Java Expressions.

In summary, using `Explore` and `Verify` we transform the original program into a metaprogram P_m. On input P_m, the JPF model checker explores the behaviour of each program variant derivable from the changeset on each generated test input.

2.4.3 Search Strategy

We discuss how to efficiently search for a correct program using the meta program P_m introduced in the last section. We modify JPF's depth-first search strategy so that it *backtracks* instead of terminating when it finds an error, thus forcing a complete exploration of all test inputs and program variants.

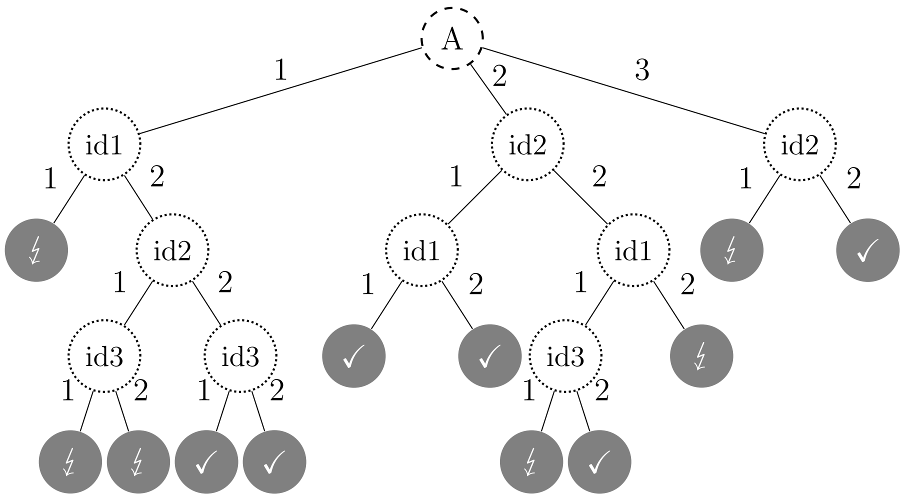


Figure 2.8: Search tree

We use a search tree to visualize the depth-first search (see Figure 2.8). Nodes are either choice points on which we branch, or end states where we start to backtrack. There are two different types of choice point nodes: those generating test inputs (TC-points, displayed as dashed nodes — "- - -"), and those where we choose different program variants (PC-points, displayed as dotted nodes — "........"). Each program execution path ends at a leaf (grey node). Two possible end states are possible: "✓"— the execution terminated without an error and "↯"— the execution terminated with an error. An error is induced by one of the following events:

- An uncaught exception occurs.
- The execution takes too long — this is necessary, because infinite executions are possible and JPF cannot detect them. We therefore restrict the amount of instructions for each path of the execution.
- The specification is violated.
- The program exits, but the defined end state of the program has not been reached.

A *program decision* is a tuple $\langle$PC-point, choice$\rangle$, like for example $\langle \text{id1}, 2\rangle$. A *decision trace* is a set of program decisions where each PC-point occurs at most once. The *size* of a trace is the number of tuples it contains. Each (partial) path in the search yields such a decision trace. A *complete decision trace* T_c contains a program decision for every defined `PC`-point, i.e. a decision for each changeset entry. It characterizes a simulated program variant $P(T_c)$ within the meta-program P_m. We search for a complete trace T_c such that $P(T_c)$ satisfies the specification on every test input.

Assume an end state is reached by depth-first search. If it is an error state, then its decision trace *cannot be* a subset of any complete trace T_c^1 such that $P(T_c^1)$ is the correct program, because $P(T_c^1)$ fails at least on one test input. If it is a success state, then its decision trace is a *candidate trace*: it could be a subset of a complete trace T_c^2 such that $P(T_c^2)$ is a correct program on all test inputs. This motivates storing two sets of decision traces during the search: a set containing the candidate traces (*good traces*), and another one containing the traces reaching an error state (*bad traces*).

Whenever an end state is reached during the depth-first search, Algorithm 4 is executed. If the end state is a success state and its decision trace does not contain a bad trace as subset, we add it to the set of good traces. If the end state is an error state, the extracted error trace can be shortened if there exists a `PC`-point in the search path towards the error, such that every successor of this node leads to an error state. This information is made available when a `PC`-point was completely explored. Therefore, whenever we hit an error state that is an end state, we do not add its trace to the set of bad traces, instead we mark the actual choice of the above `PC`-point as bad. When a `PC`-point was completely visited, we check if all its choices are marked as bad. If this is the case, we again mark the current choice of the `PC`-point above as bad. If there exists no such node above, we cannot find a correction. If not all choices are marked as bad, the decision traces of successors that induce an error state are stored as bad traces. This procedure is shown in Algorithm 5.

Consider for example the search in Figure 2.8. The path `<A,1>-<id1,2>-<id2,1>-<id3,1>` reaches an error state and choice `1` in `id3` is marked as bad. The path `<A,1>-<id1,2>-<id2,1>-<id3,2>` also reaches an error state, so now the choices `1` and `2` in `id3` are marked as bad. When backtracking, since all choices of `id3` are bad, we propagate this to `id2`, marking its choice `1` as bad. After `id2` is completely explored, not all its choices are marked as bad and so we add the decision trace {`<id1,2>`,`<id2,1>`} to the set of bad

```
Algorithm 4: FinalStateReached
input: Final state $f$, reference to a set of good traces $g$, reference
       to a set of bad traces $b$.
begin
    if $f$ is errorState then
        BackTrackToProgramChoicePoint();
        $choice\_point$.markCurrentChoiceBad();
    else
        $d_c \longleftarrow$ extractDecisionTrace();
        if $\forall\ d_b \in b$: $d_b$ does not contradict $d_c$ then
            $g$.add($d_c$);
```

decision traces.

After a bad decision trace is added, the set of good traces is updated so that all decision traces containing the new bad decision trace are removed. If all successor choices are marked bad and we have completely explored all choices of the top-most PC-point, no correct candidate decision trace for a test input was found and therefore we cannot derive a correct program and return failure.

During the search, the set of bad traces is used to prune the search space. Whenever we are about to visit a path whose decision trace contains as subset a decision trace from the set of bad traces, we skip this path, because we cannot find a good decision trace in it.

Consider again the search in Figure 2.8. Since the path `<A,1>-<id1,1>` is erroneous, the decision trace T_{e1} ={`<id1,1>`} is added to the set of bad traces after backtracking. When we explore `<A,2>-<id2,2>`, we observe that its decision trace, {`<id2,2>`,`<id1,1>`}, contains T_{e1} as subset and so we skip this path.

The search returns a set of candidate decision traces that may not be complete: for some hotspot the candidates may not indicate which changeset entry should be chosen. In this case, for each such hotspot we retain the original program expression. Each decision trace is thus extended into a complete trace. We then select one complete trace such that the number of hotspots at which the selected alternative differs from the original one is minimal. Before presenting the so obtained patch to the user, the patch is checked again for correctness, using some more sophisticated testing method.

Algorithm 5: PCPointExplored

```
input: Program choice point $p$, reference to a set of good traces
       $g$, reference to a set of bad traces $b$.
begin
    if All choices of $p$ are marked bad then
        if $\exists$ prevProgramChoicePoint(p) then
            $p\_prev \longleftarrow$ prevProgramChoicePoint($p$);
            $p\_prev$.markCurrentChoiceBad();
        else
            exit search - no solution found;
    else
        forall the $c \in$ choices marked as bad in $p$ do
            $d_b \longleftarrow$ getDecisionTrace(c);
            if $\nexists\ t \in b$ with $t \subseteq d_b$ then
                $b$.add($d_b$);
                $g$.removeContradictingDecisionTraces($b$);
```

2.4.4 Efficient Data Structure for Decision Traces

In this section we present an efficient data structure for storing a set of decision traces. Basically, a decision trace is a set of elements so this data structure holds a *set of sets of elements.* Using notation from set theory, a set of sets is referred as family of sets. This data structure allows implementing following operations efficiently:

- Add set S to (remove set S from) family of sets $\mathfrak{F}$. (Use case: $\mathfrak{F}$ is a set of decision traces, S is added/removed.)

- Extract all sets of elements from $\mathfrak{F}$ that contain S as subset. (Use case: $\mathfrak{F}$ is a family of good decision traces and S is a new bad decision trace. We remove all decision traces from $\mathfrak{F}$ containing S.)

- Check if there exists a set in $\mathfrak{F}$ that has S as subset. (Use case: S is a good decision trace if it does not contain any bad decision trace from the family of bad decision traces $\mathfrak{F}$ as subset.)

Figure 2.9 shows the representation of a family of three sets S_1-S_3. We store sets in their explicit representation together with respective set sizes (Figure 2.9/left). Additionally, we store each element in a reverse lookup map, mapping set elements to set references, stating in which sets this element occurs (Figure 2.9/right).

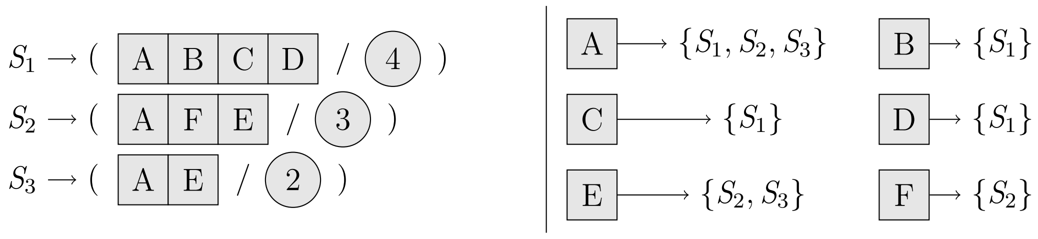

$S_1 \rightarrow$ (A B C D / 4)
$S_2 \rightarrow$ (A F E / 3)
$S_3 \rightarrow$ (A E / 2)
A $\rightarrow \{S_1, S_2, S_3\}$
B $\rightarrow \{S_1\}$
C $\rightarrow \{S_1\}$
D $\rightarrow \{S_1\}$
E $\rightarrow \{S_2, S_3\}$
F $\rightarrow \{S_2\}$

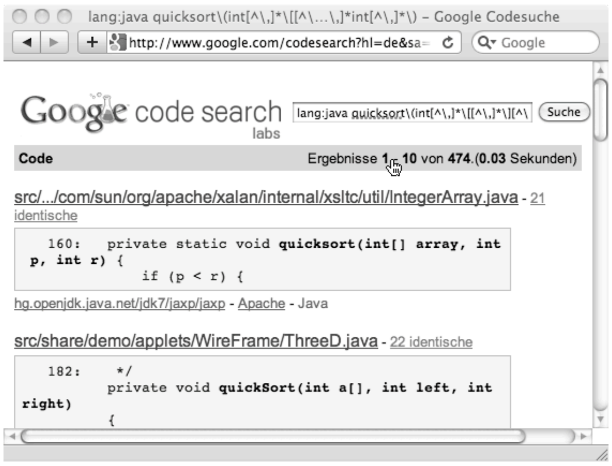
Figure 2.9: Data structure for family of decision traces

Adding/removing a set S to/from a family of sets $\mathfrak{F}$ is computationally not expensive; we have to compute the size of S and for adjusting the reverse mapping of set elements to sets we have to iterate over it once.

To compute all sets in $\mathfrak{F}$ that contain S as subset, we derive the cut set of the reverse mapping (see Figure 2.9/right) of the elements contained in S. The references to sets in the cut set are the references to the searched sets. For the example, assume $S = \{A, F, E\}$. The cut set of the three reference sets $(\{S_1, S_2, S_3\} \cap \{S_2\} \cap \{S_2, S_3\})$ contains only one element: S_2, so $S \subseteq S_2$.

For checking if S contains a set from $\mathfrak{F}$ as subset we create a mapping M, mapping each set in $\mathfrak{F}$ to a counter initialized with zero for each set. For each element in S we get a list of sets containing this element using the reverse mapping. For each set in this list we increase the counter in M of this set by one. After all elements from S are processed, we iterate through the entries in the map M. If for any map entry $\langle r, c\rangle$ (r is a set, c is its counter), the value of counter c is equal to the size of the set referenced by r, S contains at least one set from $\mathfrak{F}$ as subset, the set referenced by r.

For example, assume $S = \{A, C, D, E\}$. We apply the procedure, and we get the map $M = \{S_1 \rightarrow 3, S_2 \rightarrow 2, S_3 \rightarrow 2\}$ as interim result. The counter of S_3 matches the size of S_3, so $S_3 \subseteq S$.



lang:java quicksort\(int[^\,]*\[[^\...\,]*int[^\,]*\) – Google Codesuche

http://www.google.com/codesearch?hl=de&sa=

lang:java quicksort\(int[^\,]*\[[^\,]*\][^\

Code Ergebnisse **1** - **10** von **474**.(**0.03** Sekunden)

src/.../com/sun/org/apache/xalan/internal/xsltc/util/IntegerArray.java - 21 identische

```
private static void quicksort(int[] array, int
p, int r) {
            if (p < r) {
```

hg.openjdk.java.net/jdk7/jaxp/jaxp - Apache - Java

src/share/demo/applets/WireFrame/ThreeD.java - 22 identische

```
 */
private void quickSort(int a[], int left, int
right)
{
```

Figure 2.10: Google Code Search™

The depth-first search iteratively adds/removes (due backtracking and forward search) elements to/from a set of elements (decision trace) S, which has to be checked every time for the inclusion in some set from $\mathfrak{F}$ as subset. As discussed, this allows us to efficiently prune the search. Since this check is performed very often, it must be implemented efficiently, and so our implementation of the procedure discussed above is iterative: whenever one element of S is modified, the map M is adjusted by increasing/decreasing the respective counters.

2.5 Experiments

We demonstrate the feasibility of our approach by means of an experiment using Quicksort implementations. The implementations were obtained automatically from the web using *Google Code Search*™, a search engine for source code that supports searching for different programming languages and for regular expressions. We searched for Java implementations using typical Quicksort signatures. See Figure 2.10 for an example of manually searching for quicksort algorithms.

The relevant Quicksort methods, together with all dependencies, were fetched from the obtained implementations. A specification and an adapter was generated to supply a unique interface for executing the different sorting algorithms. Duplicates were removed, and the compilable implementations

	Domain	LOC	#Hotspots	Patch Size	Time[s]
1	framwork.googlecode.com	29	7	no fix	2
2	www.cs.iastate.edu	32	10	1	524
3	geo.jm-art.cz	29	11	6	79
4	raider.muc.edu	24	7	no fix	5
5	archive.godatabase.org	27	10	1	442
6	www.jeckle.de	22	8	no fix	158
7	www.cs.indiana.edu	35	9	3	440
8	www.cse.buffalo.edu	50	16	no fix	4
9	gwt-greflect.googlecode.com	32	11	no fix	4
10	downloads.sourceforge.net	35	11	no fix	3

Table 2.1: Experimental results on Quicksort algorithms

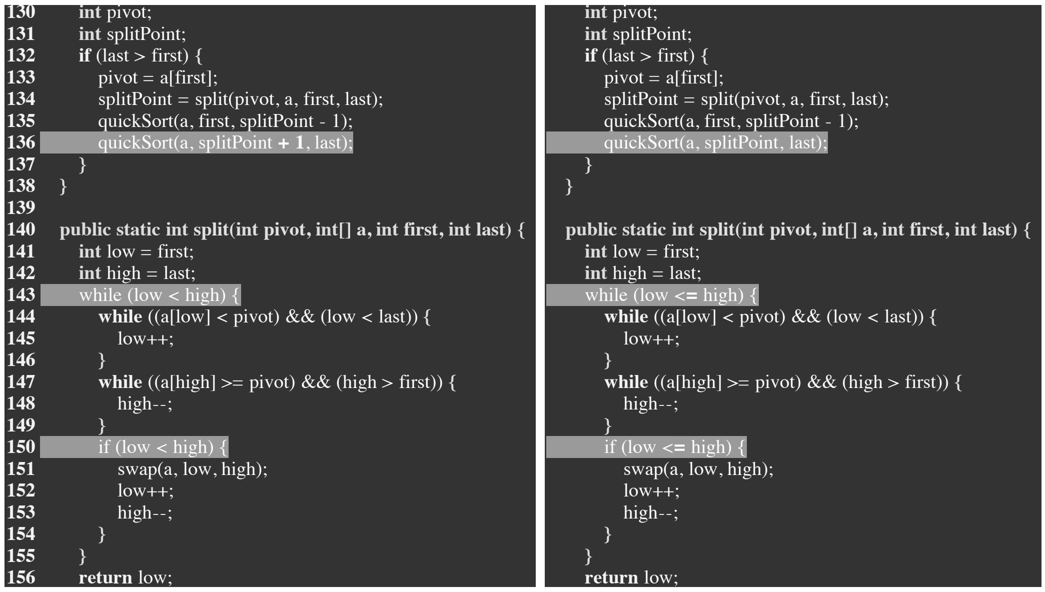
were checked automatically for correctness, which partitioned them into two categories: *correct* and *defective*. We obtained a total number of 174 source files; 88 of them were parse- and compilable and the Quicksort method was extractable with all its dependencies. Of these 88 source files, 76 were *correct* and 12 were *defective*. A closer look at the defective implementations showed that two contained an empty Quicksort method, leaving 10 *defective* implementations for our experiments.

For the search of hotspots, we focused on *off-by-one* errors, in which the bounds of a loop are wrong by one unit, or a `<=` instead of a `<` comparison is used. Therefore, our changeset C was created as follows:

- For each program location `EXPRESSION1 < EXPRESSION2`, we create the changeset entry $\textcircled{X} \rightarrow \{$ `EXPRESSION1 < EXPRESSION`, `EXPRESSION1 <= EXPRESSION2`, `EXPRESSION1 > EXPRESSION2`, `EXPRESSION1 >= EXPRESSION2`$\}$, so that each comparison operator is tried as replacement. The same is applied for all other possible comparison operators.

- For each program location `EXPRESSION1 - EXPRESSION2`, we create the changeset entry $\textcircled{X} \rightarrow \{$ `EXPRESSION1 - EXPRESSION2`, `EXPRESSION1` $\}$. The same is applied for the "+" operator.

We apply our search strategy to changeset C on 84 test inputs (all arrays of length three or less with entries between 0 and 3). We return a minimal patch when it exists.



```
    int pivot;
    int splitPoint;
    if (last > first) {
        pivot = a[first];
        splitPoint = split(pivot, a, first, last);
        quickSort(a, first, splitPoint - 1);
        quickSort(a, splitPoint + 1, last);
    }
}

public static int split(int pivot, int[] a, int first, int last) {
    int low = first;
    int high = last;
    while (low < high) {
        while ((a[low] < pivot) && (low < last)) {
            low++;
        }
        while ((a[high] >= pivot) && (high > first)) {
            high--;
        }
        if (low < high) {
            swap(a, low, high);
            low++;
            high--;
        }
    }
    return low;
```

```
    int pivot;
    int splitPoint;
    if (last > first) {
        pivot = a[first];
        splitPoint = split(pivot, a, first, last);
        quickSort(a, first, splitPoint - 1);
        quickSort(a, splitPoint, last);
    }
}

public static int split(int pivot, int[] a, int first, int last) {
    int low = first;
    int high = last;
    while (low <= high) {
        while ((a[low] < pivot) && (low < last)) {
            low++;
        }
        while ((a[high] >= pivot) && (high > first)) {
            high--;
        }
        if (low <= high) {
            swap(a, low, high);
            low++;
            high--;
        }
    }
    return low;
```

Figure 2.11: Example for an automatically generated patch

Table 2.1 shows our results. For each implementation, *Domain* denotes the domain it was fetched from, *LOC* denotes the lines of code of the extracted algorithm, *GT* and *BT* denote the number of good and bad decision traces that have been stored when the search terminates; *PC-points* denotes the number of choice points that were introduced, *Patch Size* denotes the number of code changes in the patch, and *Time* denotes the runtime of the search algorithm in seconds. Quicksort algorithms #2 and #5 are very similar, but not identical. Four out of ten Quicksort algorithms were fixed fully automatically. Figure 2.11 shows an example for such an automatically generated bugfix. All experiments were performed on an Intel Core 2 Duo 2.26GHz™ system with 3GB physical memory.

2.6 Conclusions

We have presented an approach for automatic bug fixing of Java programs that uses search techniques to explore the behaviour of program variants (candidates for a fix) on test inputs. The approach has been implemented on top of Java Pathfinder (JPF), which allows to encapsulate all program variants into one single meta-program, and using the JPF model checker to search all variants on all inputs. We have designed an efficient search strategy for early pruning unsuitable variants, and we have provided an efficient implementation with a suitable data structure.

We have tested the approach on implementations of Quicksort obtained through an automatic web search. Under the assumption that the bug was caused by "off-by-one" errors, four out of ten faulty implementations could be automatically repaired.

While the idea of exploring a set of program variants using some kind of systematic search is not new, we think that our particular design choices have two strong points. First, our search strategy makes the approach very suitable for finding fixes requiring multiple changes in different points of the code. Second, our approach fits very well the functionality offered by JPF, which greatly reduces the implementation effort and allows profiting from a very mature tool. On the other side, we require the programmer to specify the syntactic constructs where to look for bugs, and the alternative constructs that can be tried for a fix, which can be too restrictive in important cases.

Chapter 3

Diagnosis with Petri Nets

In this chapter we give an introduction to diagnosis in general and diagnosis with Petri nets in particular. Furthermore, we present related work and give preliminaries that are utilized in later chapters. Diagnosis is a very important and well studied research area. Besides the academic aspect, diagnosis is of enormous practical relevance. Mainly, the importance of diagnosis becomes visible whenever we consider systems where failures can cause huge costs or endanger the safety of people. System components can fail due to defects caused, e.g. by mechanical hardware degeneration or because of software bugs. In large distributed systems like telecommunication networks, electrical grids or manufacturing plants, it is important to detect abnormal behaviour of the system as soon as possible – even if such an abnormal behaviour of the system is locally restricted at first, it my propagate and affect the whole system or the system might completely loose its ability to operate under specified conditions. Components in critical systems, like, e.g. pilot displays in airplanes or valves in nuclear power plants must never fail.

The books [28] and [40] provide a good overview of the important research area dealing with diagnosis. SAFEPROCESS[1], the technical committee on fault detection, supervision and safety for technical processes, is an international yearly symposium of the International Federation of Automatic Control[2] (IFAC). In this thesis, we follow the terminology and notation standardised by SAFEPROCESS, which is also used in [28] and [40]. Before delving any deeper into the subject we introduce, following SAFEPROCESS,

[1]`http://www.safeprocess.es.aau.dk`
[2]`http://www.ifac-control.org`

the most important terms.

A *fault* is an unwanted or unspecified behaviour of the system, like, e.g. an electronic component that is malfunctioning because of a short circuit. Such a fault may not have any or only temporary impact on the global system stability. It may also be reversible. Perhaps, e.g. the device needs just to be rewired. If the *fault* is permanent, e.g. because it cannot be reversed, we have a *failure*, a permanent system malfunction.

Fault detection is applied to detect possible faults and to check that the detected faults are not spurious. Following fault detection, *fault diagnosis* is applied for detecting the kind and location of the fault.

Monitoring is the real-time task of observing the system and recording information that can, e.g. be used for *fault detection.* This is, e.g. realized by equipping the system with sensors that observe and register events executed by the system. These sensors emit *alarms* when potential faults are detected.

While monitoring is usually a passive task, *supervision* is the active task of monitoring a system and, if suspicious behaviour is detected, taking actions to keep the system running in an operable state. This is, e.g. achieved by automatically detecting and reversing faults. With the very general term *diagnosis* we refer to the complete task involving fault detection and fault diagnosis.

Two fundamental approaches for doing diagnosis are identified in [40]. One is having redundant components within a system. Then voting is used to determine if a component is faulty, and, if the answer is affirmative, which component is faulty. Because of the costs of additional hardware, this is a very expensive approach. Nevertheless, due to its simplicity, it is a reliable and effective approach. More relevant for us is the second approach, where analysis methods are used for doing diagnosis. For further reading, the article [42] extensively compares both approaches.

In most of the published work that can be found within the scope of the analytic approach, the system is equipped with sensors. From a system execution, data from these sensors is recorded. This recorded data is called the system *observation.* The observation is utilized for doing diagnosis. We identify two main subsidiary approaches – *symptom-based* and *model-based* methods. With symptom-based methods, the observation is directly mapped to faults or supervision actions, using rules, dictionaries, decision trees or more advanced approaches like, e.g. neural networks. See, e.g. [31] for a rule based approach and [49], [56] for neural network based approaches.

The oldest symptom-based method is probably *range checking*: directly measurable system parameters are checked for being within a certain specified range. Such a parameter could be, e.g. the range of the allowed input voltage of some electronic circuit. A system not operating within its specified normal parameters is considered faulty. See, e.g. [5], where range checking is applied to diagnose the engines of a space shuttle. Sensors for range checking are usually simple and cheap. If we, e.g. want to check whether the electric input voltage of some electronic circuit component is above or below a given trigger value, solely a voltmeter attached to the examined circuit and some alarm device indicating a range violation is needed. As being both, reliable and cheap, range checking is still very popular today.

For applying *range checking*, the value domain of the directly measurable signal has to be partitionable into ranges that classify the measured signal into two classes, "normal" and "abnormal". For complex signals, this is usually not possibly. Assume, e.g. the output voltage of an AC generator in some power plant. If we specify its normal system behaviour with respect to its output AC frequency, *range checking* cannot be applied.

This leads to *model-based* methods. The system is equipped with sensors and a model of the system is created. Then, a relation between the data of these sensors and the model is established. On the one hand, for fault detection, the acceptable system behaviour is specified within the model. The model is analysed to observe current or predicted specification violations. On the other hand, for fault explanation, behaviours of the model are derived that show the given fault and with analyzing these behaviours it is tried to locate and explain the fault within the model. Fault explanations are subsequently mapped backed to the real system. As the model is usually an abstraction of the real system, one has to carefully ensure that no spurious fault explanations are given and no spurious alarms are triggered.

The difference between these model-based methods is mainly the utilized model. We distinguish between two main model categories — discrete and continuous time system models. Both areas have been widely studied. Important representatives for the continuous case are process and signal models. Signal models abstract, e.g. electronic signals. For our previous example, when we are interested in the frequency of some complex AC signal, the Fourier Transformation is a very suitable model. See, e.g. [33], where the wavelet transformation is used for doing motor fault diagnosis. Process models are used to model relations between input and output signals of some system. In [47], e.g. neural networks are utilized as process models.

The discrete case is dominated by *discrete event system* models, DES for short. Such a model does not have notion of time, the system progresses exclusively due to the occurrence of events. The best-known DES models are finite state automata and Petri nets. Our approaches towards fault explanation are within this area. In this chapter we therefore focus on DES models and proceed as follows. In Section 3.1 we give an overview on related work regarding diagnosis of systems modelled as DES. Then, in Section 3.2, we introduce preliminaries on Petri nets and net unfoldings, where we summarize well-known notation and fundamental techniques used in later Chapters 4 and 5. In Section 3.3 of this chapter, we present and summarize the work of Stefan Haar et al. on model-based diagnosis using Petri nets and net unfoldings; Haar's approach is also the one we follow in this thesis.

3.1 Diagnosis for DES

In this section, we specifically discuss related work towards model-based diagnosis, where systems are modelled as DES. Distributed systems usually consist of many components where each single component is itself very complex. This renders observing the whole system behaviour impracticable. Systems are therefore partially observable. Usually, it is assumed that we have an executable DES model at our disposal. Progress in the model is solely achieved by the occurrence of events. In a DES, the occurrence of an event is usually modelled with firing a transition and partial observability by partitioning transitions into visible and invisible transitions. The model maps the system in such a way such that for each pair of system execution and the respective model execution it holds that the observation of the system and the sequence of visible events of the model execution are equal.

In the literature, we can identify two major problems raising in this specific area. (1) Deciding if a system is fault diagnosable and (2) applying fault diagnosis and explanation. Problem (1) raises from the fact that we consider partial observable systems. Failures may not be observable and can, e.g. propagate in such a way so that all fault symptoms are hidden, so even with infinite amount of time one can never notice the failure. We say that a system is *diagnosable* if one can always tell, in finite time, whether or not a failure has occurred. In other words, a system is diagnosable if there exist no two (infinite) runs of a system, showing the same observation, with one run containing the fault and the other one not. Our work is focused on problem (2), still approaches for problem (1) are also very closely related, so we discuss related work on both problems and additionally on work covering both areas.

Problem (1) has been widely studied. In [35] the diagnosability of circuit systems modelled as finite state automata is considered. In [43] the diagnosability problem was the first time introduced with respect to formal languages. In [53], a polynomial time algorithm is presented for deciding this problem. This is realised by creating a verifier that compares all two runs of the system for showing the same observation, one containing a fault and the other one not. The used technique is the computation of the product of the model, a finite state automaton, with itself. In [36], this work is adapted to Petri nets. The unfolding of the product of the Petri net model with itself is utilized to solve the problem. Among the works dealing with problem (1), this work is the most closest to ours, because in our techniques we also utilize Petri net unfoldings. In [19], a distributed diagnoser for Petri net models is presented that allows executing online diagnosis. A disadvantage of the construction of such a verifier or diagnoser is that its size is usually worst case double exponential in the size of the system description [20]. To tackle this state explosion, in [20] a sat based approach is presented that translates the problem into a SAT formula that can efficiently be solved by using state-of-the-art SAT solvers. We also use SAT techniques for the further analysis of the constructed fault explanation.

For problem (2), many different approaches exist that mainly differ in the used model. In [37] the situation calculus is used as model. Given a theory of system behaviour and a system observation, conjecturing diagnosis is applied to derive an explanation from the observation. Following the already mentioned work [43], Sampath et. al. present a technique [44] for creating a diagnoser which allows to apply diagnosis to Petri nets. In [3] (and related papers like [18, 17]), safe nets are used as the model and partial order techniques (unfoldings) are used to compactly represent explanations. Following this work, we also use net unfoldings as our main technique. This work is closest related to our work. Therefore, in Section 3.3, we go deeper into this work and extensively discuss its foundations. Before that, we need the knowledge of some basic notations and techniques, which are introduced in the following section, Section 3.2.

3.2 Petri Net Preliminaries

The diagnosis approaches we present in Chapters 4 and 5 utilize Petri nets and their unfoldings as model. In this section, we therefore give a common and extensive introduction on preliminaries for these topics. We summarize commonly known notation and definitions on Petri nets as well as introduce to the technique of net unfoldings, that is used for representing Petri net behaviours in a partial order compressed way. We proceed as follows. In Section 3.2.1 we introduce Petri nets, in particular safe *Petri nets* that we consider for our approach. Then, in Section 3.2.2, we introduce *occurrence nets*. Occurrence nets are, together with other properties, acyclic Petri nets. These nets are important for us, as unfoldings are occurrence nets and we need to have their properties at our disposal. Then, in Section 3.2.3, we discuss actual *unfolding techniques*, that are the base for our diagnosis techniques.

3.2.1 Petri Nets

A *Petri net* is a quadruple $N = (P, T, F, M_0)$, where P and T are disjoint sets of *places* and *transitions*, respectively. $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation* connecting transitions to places and vice versa. A function $M\colon P \to \mathbb{N}$ is called a *marking*, a function $W\colon F \to \mathbb{N}$ is called weighting and M_0 is the *initial marking*. A *node* is an element from $P \cup T$, an *arc* is an element from F. The caption of an arc is the weight of the node. An arc with no caption has weight 1. Figure 3.1 shows the graphical representation of a Petri net. We have transitions $a \dots f$ and places $1 \dots 5$. The flow relation is represented with directed arcs between nodes. For example, we have $(a, 1) \in F$, as there is an directed arc between transition a and place 1. As there is no arc in the other direction, we have $(1, a) \notin F$. As no arc has a caption, for all $f \in F$ it holds that $W(f) = 1$. The marking function is represented by the number of tokens (dots) in a place. In the figure we have, e.g. $M(5) = 0$ and $M(4) = 1$. There could also be more markings in one place.

For a node x, ${}^\bullet x := \{\, x' \mid (x', x) \in F \,\}$ is the *preset* of x and $x^\bullet := \{\, x' \mid (x, x') \in F \,\}$ is the *postset* of x. Moreover, for any set $X \subseteq P \cup T$, we define ${}^\bullet X$ and $X^\bullet$ as follows:

$$ {}^\bullet X := \bigcup_{x \in X} {}^\bullet x \quad \textit{and} \quad X^\bullet := \bigcup_{x \in X} x^\bullet $$

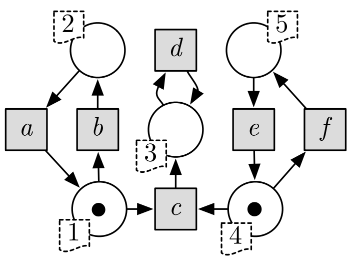


Figure 3.1: 1-safe, unweighted Petri net with transitions $a \dots f$ and places $1 \dots 5$.

In the Petri net shown in Figure 3.1, the preset of transition c is, e.g. ${}^\bullet c = \{1, 4\}$ and its postset is $c^\bullet = \{3\}$. For the preset of a set we have, e.g. ${}^\bullet({}^\bullet c) = \bigcup_{x \in ({}^\bullet c)} {}^\bullet x = ({}^\bullet 1) \cup ({}^\bullet 4) = \{a, e\}$. Shortly, we write ${}^{\bullet\bullet}c$ for ${}^\bullet({}^\bullet c)$.

Transitions induce a *firing relation* between markings; let M, M' be two markings and t a transition. We write $M \xrightarrow{t} M'$ iff for every place $p \in {}^\bullet t$ following conditions hold:

- $M(p) \geq W((p, t))$
- $M'(p) = M(p) - W((p, t))$ if $p \in {}^\bullet t \setminus t^\bullet$
- $M'(p) = M(p) + W((t, p))$ if $p \in t^\bullet \setminus {}^\bullet t$
- $M'(p) = M(p) - W((p, t)) + W((t, p))$ if $p \in t^\bullet \cap {}^\bullet t$

Loosely speaking, the first condition assures that in M, each preplace holds at least as many tokens as stated by the weight from that preplace to the transition. The other three conditions are responsible for the actual effect of "firing" t; the amount of tokens as specified by the respective "incoming" weights are removed from the preplaces of t and the respective amount of tokens as denoted by the "outgoing" weights are put into the respective post-places of t. This results in the marking M'. We say that t is *enabled* in M and that *firing* t leads to M'.

Figure 3.2 shows an example for that. On the left, a Petri net with one transition, four places and marking M is shown. Weights are given as arc labels. It is easy to see that the first condition for firing transition t' holds for all preplaces. We "apply" the three other conditions and obtain marking M', shown right of the figure.

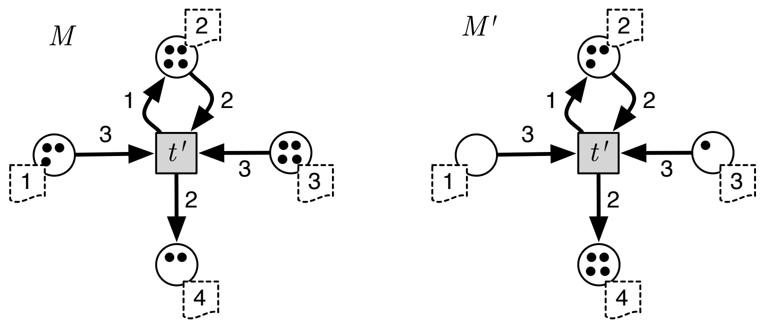


Figure 3.2: Firing of a transition

A finite sequence $\sigma := t_1 \dots t_k$ of transitions is a *run* iff there exist markings $M_1, \dots, M_k$ so that $M_0 \xrightarrow{t_1} M_1 \cdots \xrightarrow{t_k} M_k$, where M_0 is the initial marking of the net. If such a run exists, then M_k is said to be *reachable.* The set of reachable markings is denoted by $\mathbf{R}(N)$.

A Petri net is k-safe iff no reachable marking puts more than k tokens into any place. A Petri net is said to be unweighted, if $W(a) = 1$ for all arcs a. All Petri nets we are interested in are 1-safe and unweighted, for now on we may call them shortly (1-safe) nets. Figure 3.1 is such a 1-safe and unweighted net. For shorter notation, a marking in a 1-safe Petri net is represented as set $M^{-1}(1)$. In the net from Figure 3.1, we have $M_0 = \{1, 4\}$. There exists a run $\sigma_1 = baf$ resulting in reachable marking $M_{\sigma_1} = \{1, 5\}$. Marking $M = \{1, 3\}$ is not reachable.

It is a PSPACE-complete problem to decide whether or not a given (unweighted) Petri net is 1-safe, however one can easily guarantee 1-safeness by construction [4]. Many interesting problems can be formulated using 1-safe Petri nets as, e.g. the reachability problem in synchronized products of transition systems. Moreover, most results obtained from 1-safe Petri nets also hold in the general case.

An infinite sequence of transitions $t_1 t_2 \dots$ is called a *run* if every prefix of it is one. We say that a run σ is *fair* iff one of the following holds:

- σ is finite, and in the marking reached by σ, no transition is enabled
- $\sigma = t_1 t_2 \dots$ is infinite, where $M_1, M_2, \dots$ are the markings generated by firing σ, and there exists no pair $t \in T$ and $i \geq 1$ such that t is enabled in all M_k, $k \geq i$ and $t \neq t_k$ for all $k > i$.

In other words, a fair run cannot delay firing an enabled transition forever. For example the run $\sigma_1 = bfea(bafe)^\omega$ is a fair run whereas $\sigma_2 = bfea(ba)^\omega$

is no fair run, because transition f is enabled forever, after firing the third transition.

3.2.2 Occurrence Nets

In this section, we introduce occurrence nets, a restricted version of nets. We are interested in their properties, as the unfolding of a net is an occurrence net. Therefore, we mainly summarize well-known facts, e.g. given in [13], [15].

An occurrence net is a restricted version of a net. It is therefore also a quadruple. The components however are renamed; places to *conditions* and transitions to *events*. This is explainable, as the net is acyclic and so each transition can at most fire once in a run. So a transition is an event and its preplaces are the conditions for this event to happen. Moreover, this renaming helps to avoid confusion between transitions in the net and transitions of its unfolding. We have occurrence net O with $O = (C, E, F, C_0)$.

Let $a < b$ for two nodes a and b iff $(a, b) \in F$ and $a \leq b$ iff $a = b$ or $a < b$. Furthermore, let $<$ denote the transitive closure F^+ and $\leq$ the reflexive and transitive closure F^*. An acyclic net is a net where $\leq$ is a partial order.

We define three additional relations on the nodes in acyclic nets (with their respective symbols): *conflict* relation($\#$), *concurrency* relation($\mathbf{co}$) and *causal* relation ($\rightleftharpoons$). Fix two nodes $a, b \in C \cup E$, then

- $a \rightleftharpoons b$ iff $a \leq b$ or $b \leq a$
- $a\#b$ iff there exist $e, e' \in E$ such that *(i)* $e \neq e'$, *(ii)* ${}^\bullet e \cap {}^\bullet e' \neq \varnothing$, and *(iii)* $e \leq a$ and $e' \leq b$
- a $\mathbf{co}$ b iff $\neg(a \rightleftharpoons b)$ and $\neg(a\#b)$

Given an event e, we denote with $\#[e]$ the set of events in conflict with e (analogous $\mathbf{co}[e]$ and $\rightleftharpoons [e]$). A $\mathbf{co}$-set is a set of nodes pairwise in concurrency relation (analogous for $\#$ and $\rightleftharpoons$). Furthermore, let $\lceil e \rceil := \{\, e' \in E \mid e' \leq e \,\}$ be the *cone* of e, and $\lfloor e \rfloor := \lceil e \rceil \setminus \{e\}$ the *precone* of e. We now define occurrence nets.

Definition 1 (Occurrence net)**.** *An occurrence net is an acyclic net, that satisfies following properties:*

- *no self conflict:* $\forall\, x \in C \cup E \colon \neg(x \# x)$*;*
- *finite cones: all events* e *satisfy* $|\lceil e \rceil| < \infty$*;*
- *no branching on conditions: all conditions* c *satisfy* $|^\bullet c| \leq 1$*;*
- $C_0 \subseteq C$ *is the set of* $\leq$*-minimal nodes.*

Note that in occurrence nets, for all pairs (a, b) of nodes exclusively one of the three presented relations (conflict, causality or concurrency) holds. As the "initial marking" of an occurrence net is uniquely determined by the set of its minimal conditions with respect to $\leq$, the tokens of the initial marking are usually omitted in its graphical representation. Figure 3.3 shows an occurrence net. The initial marking is $C_0 = \{1, 2\}$. Nodes d and f are both in conflict with b (d and b compete for condition 1, f and b for condition 2), yet not with each other. Moreover, d and f are not in causal relation (neither $d \leq f$ nor $f \leq d$ holds), so d and f are in concurrency relation.

We now define the *prefix* of an occurrence net. Remember from set theory that a set is downward closed, iff for every element all smaller elements with respect to $\leq$ are contained. Informally, a prefix is a downward closed "part" of the occurrence net that contains all initial conditions as well as retains the flow relation between contained nodes.

Definition 2 (Prefix)**.** *Let* $O = (C, E, F, C_0)$ *be an occurrence net. A* prefix *of* O *is an occurence net* $O' = (C', E', F', C_0)$*, with* $C' \subseteq C$*,* $C_0 \subseteq C'$*,* $E' \subseteq E$ *and* $F = (C' \cup E')^2 \cap F$*, that fulfils following properties:*

- *If* $x \in (C' \cup E')$*, then* $\forall\, y \in (C \cup E), y < x : y \in (C' \cup E')$*.*
- *If* $x \in E'$*, then* $\forall\, y \in x^\bullet, y \in C'$*.*

The first property assures that the prefix is downward closed. The second property denotes that each event in the prefix contains all its post-conditions. We write $O' \sqsubseteq O$. Note that such a prefix is uniquely determined by its set of events. A prefix is called finite if C' and E' are finite sets. We denote with $O[E']$ the unique prefix of O, whose set of events is E'.

We say that a set S of events is conflict free if it is pairwise conflict free. More formal: forall pairs $(a, b) \in (S \times S)$, it holds that $\neg(a\#b)$. We now define a *configuration.* Loosely speaking, a configuration encapsulates a set of behaviours of the net, reaching the same marking.

Definition 3 (Configuration). *A configuration $\mathcal{C}$ of occurrence net $O = (C, E, F, C_0)$ is a downward closed and conflict free set of events.*

Note that in an occurrence net, the cone $\lceil e \rceil$ and precone $\lfloor e \rfloor$ of every event e are configurations. This is because of the fact that event e is, by definition of an occurrence net, not in conflict with itself and its precone is downward closed. We say that $\lceil e \rceil$ is the *local configuration* of e. Note that $\lceil e \rceil$ and $\lfloor e \rfloor$ are finite configurations.

Figure 3.3 shows an occurrence net. The set of events $\mathcal{C}_1 = \{a, d, c\}$ is a configuration, whereas $S_1 = \{a, d, b\}$ is not a configuration as, e.g. $a\#b$ holds. Moreover, the set of events $S_2 = \{a, d, f\}$ is conflict free but it is not a configuration as it is not downward closed.

As each configuration $\mathcal{C}$ is a conflict free and downward closed set of events it follows that there exists a run $\sigma_{\mathcal{C}}$ containing all those events. Moreover, the causal order of these events in $\mathcal{C}$ gives a partial order on its events and each of its linearizations is a run. Vice versa, each run in the occurrence net is a linearization of the partial order of some configuration of the occurrence net. For example, in Figure 3.3, *cad* is a linearization of $\mathcal{C}_1$, which is a run.

We now define the *cut* of a configuration. All runs of a configuration $\mathcal{C}$ "reach" the same marking $Mark(\mathcal{C})$. The cut of $\mathcal{C}$ is this marking. Formally:

Definition 4 (Cut). *Given a configuration $\mathcal{C}$, $Cut(\mathcal{C})$ is the set of $\leq$-maximal conditions of $O[\mathcal{C}]$.*

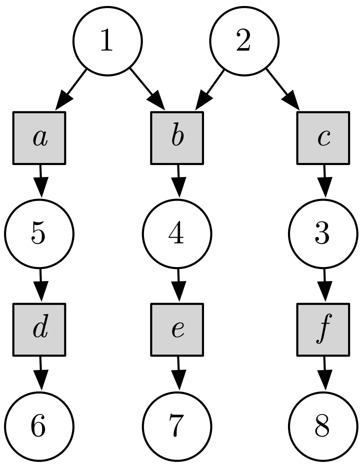


Figure 3.3: Occurrence net

Informally, the *postfix* of a finite configuration is the occurrence net we obtain by removing all nodes of the net "above" the cut of this configuration.

Definition 5 (Postfix). *The* postfix $O/_{\mathcal{C}}$ *of configuration* $\mathcal{C}$ *is the occurrence net* $O/_{\mathcal{C}} = (C'', E'', F'', C_0'')$*, where* $C'' = C \setminus {}^\bullet\mathcal{C}$*,* $E'' = E \setminus \mathcal{C}$*,* $F'' = F \cap (C'' \cup E'')^2$*, and* $C_0'' = Cut(\mathcal{C})$*.*

We now define the *extension of a configuration.*

Definition 6 (Extension of Configuration). *If* $\mathcal{C}$ *is a finite configuration and* e_{ext} *is an event so that* $\mathcal{C} \cup \{e_{ext}\}$ *is a configuration then* e_{ext} *is an extension of configuration* $\mathcal{C}$*.*

We write $\mathcal{C} \overset{e_{ext}}{\leadsto}$ or $\mathcal{C} \overset{e_{ext}}{\leadsto} \mathcal{C}'$ for that, where $\mathcal{C}'$ is the configuration $\mathcal{C} \cup \{e_{ext}\}$. Expanded to multiple events, given a set of events $A = \{e_1, \ldots, e_n\}$, we denote with $\mathcal{C} \overset{A}{\leadsto} \mathcal{C}'$ or $\mathcal{C} \overset{A}{\leadsto}$ that there exists a permutation $P = \langle e'_1, \ldots, e'_n \rangle$ of the elements of A such that there exists a set of configurations $\{\mathcal{C}_1 \ldots \mathcal{C}_{n-1}, \mathcal{C}'\}$ so that $\mathcal{C} \overset{e'_1}{\leadsto} \mathcal{C}_1 \overset{e'_2}{\leadsto} \mathcal{C}_2 \ldots \mathcal{C}_{n-1} \overset{e'_n}{\leadsto} \mathcal{C}'$ holds. Informally, A is and extension to $\mathcal{C}$ if adding all events contained in A to $\mathcal{C}$ in some order yields a configuration. We write $\mathcal{C} \sqsubseteq \mathcal{C}'$ if there exists a set A such that $\mathcal{C} \overset{A}{\leadsto} \mathcal{C}'$.

If $\mathcal{C}$ is a finite configuration and $e'_{ext} \in E \setminus \mathcal{C}$ an event such that ${}^\bullet e'_{ext} \subseteq Cut(\mathcal{C})$ then e'_{ext} is a configuration because $Cut(\mathcal{C})$ is a **co**-set and so e'_{ext} is enabled by $Mark(\mathcal{C})$. So, e'_{ext} is an extension of configuration $\mathcal{C}$. Note that from $\mathcal{C} \overset{x}{\leadsto}$ and $\mathcal{C} \overset{y}{\leadsto}$ we cannot derive $\mathcal{C} \overset{x \cup y}{\leadsto}$ as ${}^\bullet x \cap {}^\bullet y$ might be non-empty.

3.2.3 Branching Process and Unfolding

A labelled occurrence net is called a *branching process* [9]. Branching, because we branch on conditions for representing conflicts. The unique branching process, encapsulating all behaviours of the Petri net, is called *unfolding.*

The unfolding of a net is, loosely speaking, the "unrolled" net. It is an acyclic, partial order compressed, view on the net. The unfolding is usually infinite (iff the net contains cycles). This section presents well-known techniques how to (efficiently) create this unfolding and useful finite prefixes of it.

Given Petri net $N = (P, T, F', M_0)$ we construct a unfolding $U = (O, f)$, i.e. a tuple consisting of an occurrence net $O = (C, E, F, C_0)$ and a labelling function $f \colon (C \cup E) \to (P \cup T)$. Sometimes, for easier notation, we refer directly to the occurrence net ($U := O$) and just keep in mind that it is actually a tuple additionally holding the labelling function f.

Let $S_1 \subseteq (C \cup E)$ and $S_2 = (P \cup T)$. With the formalism $f\colon S_1 \leftrightarrow S_2$ we denote that $f|_{S_1}$ (the domain f restricted to S_1) is a bijection, i.e. for each element y with $y \in S_2$ there exists at most one element $x \in S_1$, so that $f(x) = y$. We are now able to formally define the unfolding of a Petri net.

Definition 7 (Unfolding)**.** *An occurrence net $O = (C, E, F, C_0)$ with mapping $f\colon (C \cup E) \to (P \cup T)$ is the unfolding of $N = (P, T, F', M_0)$ if it is the maximal branching process, so that following conditions hold:*

- $f\colon C_0 \leftrightarrow M_0$
- *for every $e_1, e_2 \in E$, if $^\bullet e_1 = {}^\bullet e_2$ and $f(e_1) = f(e_2)$, then $e_1 = e_2$ and*
- *for every $e \in E$ it holds that $f\colon {}^\bullet e \leftrightarrow {}^\bullet f(e)$ and $f\colon e^\bullet \leftrightarrow f(e)^\bullet$.*
- *for every $c \in (C \setminus C_0)$, $^\bullet c \neq \varnothing$*

Loosely speaking, the first condition assures that for each initial marking we have exactly one copy of it in the unfolding. The second condition assures that for every set of conditions we "attach" at most one copy of a matching transition. The third condition fixes the structure of the occurrence net, i.e. the labels of the pre-conditions and post-conditions of an event are the post-places and pre-places of the respective transition in the Petri net. The last condition assures that we have no conditions floating around. Each condition except the initial ones have to have a pre-event.

Note that, as occurrence nets forbid self-conflicts (Definition 1), for every e it holds that the conditions in $^\bullet e$ are pairwise in **co**-relation, otherwise, for two events $e_1, e_2 \in {}^\bullet e$ it holds that $e_1 \# e_2$ or $e_1 \rightleftharpoons e_2$ and in both cases, e is in self-conflict. Excluding self-conflicts from the unfolding makes sense, as those events would never be enabled by some configuration. The other conditions of the occurrence net solely hold due to the rules of Definition 7.

Figure 3.4 shows a finite prefix of the unfolding of the net given in Figure 3.1. The labelling function is simple, only the index has to be dropped, e.g. $f(b_2) = b$.

We now establish the connection between a net P and its unfolding U, briefly summarizing the results given in [13, 15]. U mimics the behaviour of P as follows. With every configuration $\mathcal{C}$ of U we associate the marking

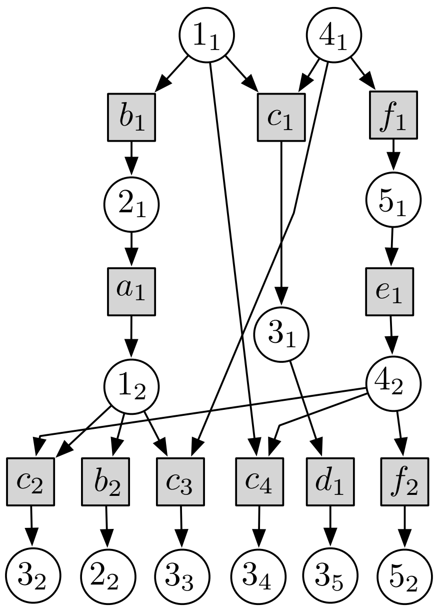


Figure 3.4: Unfolding example

$Mark(\mathcal{C}) := \{f(c) \mid c \in Cut(\mathcal{C})\}$. It is well-known [13],[15] that M is a reachable marking in N iff there exists a configuration $\mathcal{C}$ of U such that $Mark(\mathcal{C}) = M$. Moreover, if σ is a run corresponding to $\mathcal{C}$, then $f(\sigma)$ leads from M_0 to M in N. It is in this sense that U mimics the behaviour of N.

It is also well-known that for any configuration $\mathcal{C}$, the postfix $U/_{\mathcal{C}}$ is isomorphic to the unfolding of the net $(P, T, F, Mark(\mathcal{C}))$. A prefix U' of U is called *complete* if it "contains" every marking of N. For deciding if a marking is reachable, it is sufficient to examine a complete prefix. More formally:

Definition 8 (Complete Prefix)**.** *A* complete prefix *of net N with unfolding U is a prefix U_p of U such that for every marking M of N there exists a configuration $\mathcal{C}_M$, such that $\mathcal{C}_M \in U_p$ and $Mark(\mathcal{C}_M) = M$.*

We briefly discuss how to efficiently construct a finite and complete prefix, presenting the results of the work pioneered in [38], refined and continued in [15]. Let U be the unfolding of net N and U' some prefix of U. We now define the adequate order, which gives a partial order on configurations. This order is used in the unfolding procedure.

Definition 9 (Adequate Order). *We call $\prec$ an* adequate order *on configurations of U if $\prec$ is a well-founded partial order that satisfies following conditions:*

- *For all configurations C_1, C_2 with $C_1 \subset C_2$, it holds that $C_1 \prec C_2$.*
- *Let f be the mapping from Definition 7. If $\mathcal{C}_1, \mathcal{C}'_1, \mathcal{C}_2, \mathcal{C}'_2$ are configurations, A and B sets of events with $f(A) = f(B)$ so that $\mathcal{C}_1 \overset{A}{\rightsquigarrow} \mathcal{C}'_1$, $\mathcal{C}_2 \overset{B}{\rightsquigarrow} \mathcal{C}'_2$ and $\mathcal{C}_1 \prec \mathcal{C}_2$, then it holds that $\mathcal{C}'_1 \prec \mathcal{C}'_2$.*

Let e be some event. We write M_e shortly for $Mark(\lceil e \rceil)$. We now define cut-off events, i.e. events that are not expanded further in the unfolding procedure.

Definition 10 (Cuf-off event). *We call e a cut-off event, if there exists another event e', so that $e' \prec e$, with $M_e = M_{e'}$.*

Event e is said to be a *possible extension* of U' ($e \in \mathbf{pe}(U')$) if there exists a configuration $\mathcal{C}$ of U', such that $\mathcal{C} \overset{e}{\rightsquigarrow}$. Algorithm 6 constructs a finite complete prefix. We start with a prefix U_p that contains for each place $p \in M_0$ exactly one fresh condition c, so that $f(c) = p$, and nothing else. Iteratively we search for possible extensions for this prefix, and add events to it in the order given by $\prec$. With "adding an event to U_p", we refer to adding this event with all its post-conditions. Cut-off events are not further extended.

It is easy to verify that $\prec := <$ defines an adequate order. Using this order to construct a complete prefix, we can simplify our algorithm and remove the restriction, to always only add an event minimal w.r.t. $\prec$, as it is, by construction, not possible to add event x before y, if $y < x$.

3.3 Diagnosis with Unfoldings

In our work we follow the fault diagnosis approach presented by Haar et al. in [3] and related papers like [18, 17]. In this section we extensively discuss the foundations of this closely related approach. This work deals with fault explanation using partially observable labelled Petri nets as model. Informally, a labelled net is a net with an additional mapping λ from places and transitions to labels.

Algorithm 6: Compute complete prefix

```
input  : Net $N = (P, T, F, M_0)$
output: Finite complete prefix $U_p$
begin
    $U_p \longleftarrow (C = \text{copy of } M_0, E = \varnothing, F = \varnothing, C_0 = C)$;
    $PEs \longleftarrow \mathbf{pe}(U_p)$;
    $Cuts \longleftarrow \varnothing$;
    while $PEs \neq \varnothing$ do
        $e \longleftarrow$ event from $PEs$, so that $e$ is minimal w.r.t. $\prec$;
        if $e$ is cut-off then
            if $e \notin Cuts$ then add $e$ to Cuts;
        else
            Add $e$ to $U_p$;
        $PEs \longleftarrow \mathbf{pe}(U_p)$;
    Add all elements from $Cuts$ to $U_p$;
    return $U_p$
```

Definition 11 (Labelled net)**.** *A* labelled net *is a tuple* $N = (P, T, F, \lambda, M_0)$*, where* P, T, F *and* M_0 *are defined as in Section 3.2.1 and* $\lambda\colon P \cup T \to L$ *is a labelling function that assigns to each place and transition a label from alphabet* L*.*

For modelling the partial observability, the transitions of the labelled net are partitioned into visible and invisible transitions. Invisible transitions are labelled with the empty label ϵ. From a run σ of some labelled net, the observer records the sequence $O[\sigma]$, the labels of visible transitions in order of their occurrence in the run σ. Note that this sequence of visible labels corresponds to the sequence of events, the observer records from sensors of the real system. We demonstrate the approach by means of an example, as shown in Figure 3.5. Subfigure (I) shows a labelled net N with three transitions. For simplicity, only transition labels and not transition names are shown. Assume some concrete system execution, i.e. a run σ, where we first observe that transition a and then transition b is fired. Further, assume this firing triggers an alarm, so we now apply diagnosis on this possible failure.

The approach towards diagnosis is as follows. In a first step, the observa-

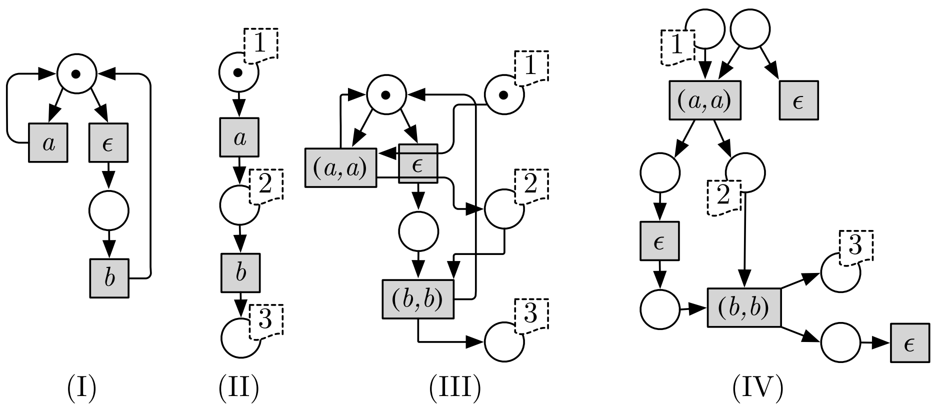


Figure 3.5: From left to right: considered system model, observation as linear net, product of the first two, unfolding of the product.

tion is transformed into a non-branching linear net, containing a transition for each observed label, carrying the same label. In our running example, we have $O[\sigma] = \langle a, b \rangle$ and the respective linear net $N_{O[\sigma]}$ is shown in Figure 3.5/(II). In a next step we compute the synchronized product $N_p = N_{O[\sigma]} \times N$, which is, for our running example, shown in Figure 3.5/(III), and, formally defined as follows.

Let $\lambda(T)$ be the set of transition labels of all transitions in transition set T and let $\lambda_T^{-1}(l)$ be the set of transitions in T with label l.

Definition 12 (Product of two labelled partial observable nets)**.** *The product $N'' = N \times N'$ of two nets $N = (P, T, F, \lambda, M_0)$ and $N' = (P', T', F', \lambda', M_0')$ is a tuple $N'' = (P'' := P \cup P', T'', F'', \lambda'', M_0'' := M_0 \cup M_0')$ so that*

1. *For all $t \in T \cup T'$ with $\lambda(t) = \epsilon$, we have $t \in T''$, and for all $t \in T \cup T'$ with $\lambda(t) \notin \lambda(T) \cap \lambda(T')$, we have $t \in T''$.*

2. *For all $t \in T''$ from Rule 1: for all $p \in {}^\bullet t$ we have $(p, t) \in F''$ and for all $p \in t^\bullet$ we have $(t, p) \in F''$.*

3. *For all labels $l \in (\lambda(T) \cap \lambda(T')) \setminus \epsilon$, and each $t' = (t_1, t_2) \in (\lambda_T^{-1}(l) \times \lambda_{T'}^{-1}(l))$: let t' be a new transitions. We have $t' \in T''$.*

4. *For all transitions $t' = (t_1, t_2)$ with $t' \in T''$ from Rule 3: for all $p \in ({}^\bullet t_1 \cup {}^\bullet t_2)$ we have $(p, t') \in F''$ and for all $p \in (t_1{}^\bullet \cup t_2{}^\bullet)$ we have $(t', p) \in F''$.*

Nothing else is in F and T.

Informally, Rule 1 denotes that all transitions whose label is only contained in one transition system or whose label is ϵ are contained in T'', and, Rule 2 denotes that these transitions retain their connections to post- and pre-conditions. With Rule 3, we add for each pair of transitions that originate from different transition systems and carry the same label a "combined transition", and, with Rule 4, these combined transitions retain the connections to post- and pre-conditions of both original transitions.

In our running example we have two combined transitions. For simplicity, we write $(\lambda(t_1), \lambda(t_2))$ instead of (t_1, t_2) (as we already do for normal transitions), so we have transitions (a, a) and (b, b). The *explanation* is the set of firing sequences of the model, showing the same observation as the system execution. It can be constructed using the product: the set of explanations are the firing sequences of the product reaching the last place of the linear net (Place 3 in the example), where on combined transitions we project onto the first element. In the example, there exists only one such firing sequence, namely $\langle (a, a), \epsilon, (b, b) \rangle$ that, projected, is the firing sequence $\langle a, \epsilon, b \rangle$. So in this example, we only have one explanation for the observation, that is the system run. The product is not a good symbolic representation for the explanation, as it has to be computed from the product by enumerating all firing sequences marking Place 3. In addition, explicitly representing this set of runs is involved, as its size explodes.

In [38], an approach was pioneered for using net unfoldings to represent sets of firing sequences in a compact, partial order compressed way. Therefore, the unfolding of the product is used to represent the explanation. For our running example, the unfolding of $N_p = N_{O[\sigma]} \times N$ is shown in Figure 3.5/(IV).

Although this is a very reasonable approach for doing diagnosis in petri nets, we identify following weaknesses:

1. *No infinite loops on invisible events.* See Figure 3.6 for an example, where the invisible transition now can occur infinitely often in a firing sequence. The unfolding of the product therefore is also infinite. Only considering complete finite prefixes, as considered in [16], is no solution, as we are not interested in reachable markings, but "reachable" sequences. We present cut-off mechanisms, for generating finite prefixes also for this case.

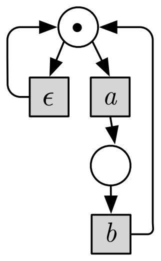


Figure 3.6: Possible loop on invisible transitions

2. *The observer is always behind.* Iteratively, the observer can record labels and the nets (including the unfolding) are extended, to be "compatible" with the extended observation. Still, we are always behind the system, not able to predict alarms/failures in the future of the system. We present an approach, for proactively looking into the future of the system and use "garbage collection" to stay compatible with the observation.

3. *No satisfying implementation.* The presented approach introduces algorithms, for solving the diagnosis problem, but no experimental implementation is given. Additionally to algorithms for solving the diagnosis problem, we also report on a very efficient implementation.

Chapter 4

Reactive and Proactive Diagnosis

In this chapter, we present a novel approach for doing diagnosis with Petri nets. We consider synchronized products of transition systems as model. Such models are very well suited, e.g. for modelling communicating processes and can be represented as Petri nets. We present algorithms for solving the diagnosis problem using these models. We are interested in the question if diagnosis can be done efficiently while the system under diagnosis is running. Unfolding prefixes are used for encapsulating the system's behaviour, compatible with the observation of the system. Using different levels of over-approximations, we enable an efficient computation of this prefix. We use SAT solving methods for explaining the behaviour and compensating the inaccuracy caused by the approximation. Furthermore, in proactive diagnosis, we not only consider compatible behaviour of the system, but also speculate about the future of the system. We proceed as follows. In Section 4.1 we present products of transition systems. Their Petri net representation is discussed in Section 4.2. Then, in Section 4.3, we discuss and evaluate our approach using a case study. Finally, we conclude in Section 4.4.

4.1 Products of Transition Systems as Model

With our approach, we utilize synchronized products of transition systems as system models. Such models are motivated by communicating processes in multi-threaded environments. In such an environment, each single thread is modelled as a finite automaton and the communication of these automatons is modelled as synchronizations between transitions of these automatons.

Such models are also very well suited for modelling systems, where different agents usually operate independently but need to be synchronized from time to time, like, e.g. warehouse robots. Such products can be described as Petri nets, therefore we can adapt to and follow the approach that we discussed in Section 3.3. Moreover, we can exploit specializations of this model (compared to Petri nets), for developing an efficient diagnosis procedure. We begin with formally defining labelled transition systems as follows.

Definition 13. *(Labelled transition system) A* labelled transition system *is a tuple $\mathcal{A} = \langle S, T, \alpha, \beta, is, \ell \rangle$, where*

- *S is a finite set of* states*.*
- *T is a finite set of* transitions*.*
- *Functions $\alpha, \beta \colon T \to S$ associate to each transition its* source *and* target *state, α and β, respectively.*
- Initial state *$is \in S$.*
- *$\ell \colon T \to L \cup \{\epsilon\}$ is a* labelling function *assigning to each transition an element from a set L of labels, or a special label ϵ, called the "empty label".*

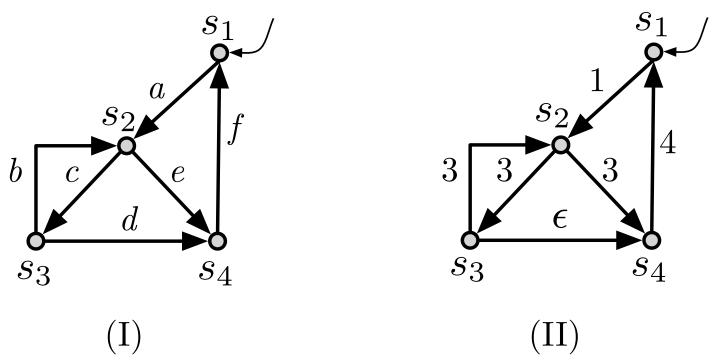


Figure 4.1: Left: transition system with transition names, Right: transition system with transition labels

Figure 4.1 shows labelled transition system $\mathcal{A}_1 = \langle S_1, T_1, \alpha_1, \beta_1, is_1, \ell_1 \rangle$, with $S_1 = \{s_1, s_2, s_3, s_4\}$, $T_1 = \{a, b, c, d, e, f\}$, functions α_1 and β_1: $\alpha_1(a) =$

s_1, $\alpha_1(c, e) = s_2$, $\alpha_1(d, b) = s_3$, $\alpha_1(f) = s_4$, $\beta_1(f) = s_1$, $\beta_1(a, b) = s_2$, $\beta_1(c) = s_3$, $\beta_1(d, e) = s_4$, initial state $is_1 = s_1$ and labelling function ℓ_1: $\ell_1(a) = 1, \ell_1(b, c, e) = 3, \ell_1(d) = \epsilon, \ell_1(f) = 4$. Subfigure I shows this system with its states S_1 represented as circles and the transitions T_1 represented as arrows, with arrow heads and tails denoting the target β_1 and source α_1 functions respectively. The arrow without a source at its tail points to the initial state of the system. Subfigure II again shows this transition system, but instead of transitions $t \in T_1$, transition labels $\ell_1(t)$ are shown.

We now introduce the *synchronized product* of transition systems, which is defined using multiple transition systems. Assume therefore a set of transition systems $\mathcal{A}_1, \ldots, \mathcal{A}_n$. For each transition system $\mathcal{A}_i$ with $i \in \{1 \ldots n\}$, we have $\mathcal{A}_i = \langle S_i, T_i, \alpha_i, \beta_i, is_i, \ell_i \rangle$. Let all the S_i's and T_i's be pairwise disjoint.

To continue our running example, in addition to transition system $\mathcal{A}_1$, we introduce transition system $\mathcal{A}_2 = \langle S_2, T_2, \alpha_2, \beta_2, is_2, \ell_2 \rangle$ with $S_2 = \{s_5, s_6\}$, $\alpha_2(g) = s_6$, $\alpha_2(h) = s_5$, $\beta_2(g) = s_5$, $\beta_2(h) = s_6$, $is_2 = s_6$, $\ell_2(g) = 2$ and $\ell_2(h) = \epsilon$. Both transition systems are shown in Figure 4.2.

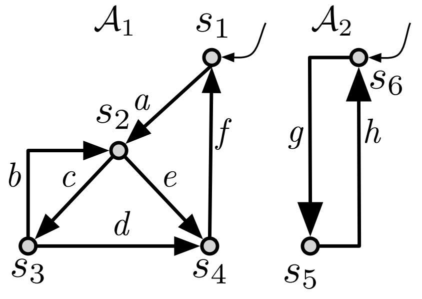


Figure 4.2: Transition systems with state labels

A *synchronization constraint* is a set of constraints where a single constraint connects a set of transitions in such a way so that no two transitions are from the same transition system. More formally:

Definition 14. *(Synchronization constraint) Let $T_1 \ldots T_n$ be the respective sets of transitions contained in transition systems $\mathcal{A}_i \ldots \mathcal{A}_n$. Further let "$*$" be a new symbol denoting idling. A* synchronization constraint $\mathbf{T}$ *is a subset of*

$$(T_1 \cup \{*\}) \times \cdots \times (T_n \cup \{*\}) \setminus \{(*, \ldots, *)\}$$

The elements of $\mathbf{T}$ are called *global transitions*. We denote the i-th component of $\mathbf{t} \in \mathbf{T}$ by t_i, i.e. $\mathbf{t} = (t_1, \ldots, t_n)$. If $t_i \neq *$ we say that $\mathcal{A}_i$ *participates* in $\mathbf{t}$. Otherwise we say that $\mathcal{A}_i$ does not participate in $\mathbf{t}$.

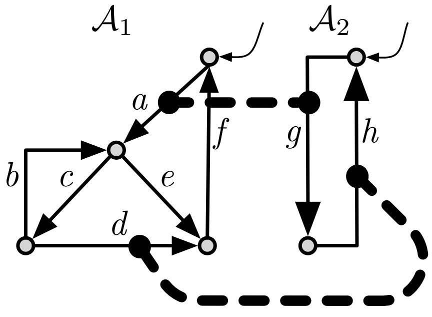


Figure 4.3: Transition systems with synchronization constraint

For our running example, we take the synchronization constraint $\mathbf{T} = \{(a,g),(d,h)(b,*),(c,*),(e,*),(f,*)\}$. This is illustrated in Figure 4.3. State names are omitted. The dashed lines that connect arrows represent synchronization constraints, where both transition systems participate, e.g. (a,g). Arrows that are not connected represent constraints, where only the transition system of the arrow participates and the other transition systems are idling, e.g. $(b,*)$. A synchronized product of transition systems is, loosely speaking, a set of labelled transition systems with a synchronization constraint. More formally:

Definition 15. *(Synchronized product of labelled transition systems) A synchronized* product *of labelled transition systems $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is a tuple $\mathbf{A} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n, \mathbf{T} \rangle$ where $\mathbf{T}$ is a synchronization constraint for the contained set of transition systems.*

A *global state* of $\mathbf{A}$ is a tuple $\mathbf{s} = (s_1, \ldots, s_n)$ where $s_i \in S_i$. The *initial global state* is the tuple $\mathbf{is} = (is_1, \ldots, is_n)$.

In our running example, e.g. (s_1, s_6) is a global state. Observe that this is also the initial global state of the system, as $is_1 = s_1$ and $is_2 = s_6$.

A *step* of product $\mathbf{A}$ is a triple $(\mathbf{s}, \mathbf{t}, \mathbf{s}')$, where $\mathbf{s} = (s_1, \ldots, s_n)$ and $\mathbf{s}' = (s'_1, \ldots, s'_n)$ are global states and $\mathbf{t}$ is a global transition such that if $s'_i = \beta_i(t_i)$ and $s_i = \alpha(t_i)$ if $t_i \neq *$, and $s'_i = s_i$ otherwise.

In our running example we have, e.g. the steps $((s_1, s_6), (a, g), (s_2, s_5))$ and $((s_3, s_6), (b, *), (s_2, s_6))$, however the later step can never occur as the global state (s_3, s_6) is not reachable.

We say that $\mathbf{s}$ *enables* $\mathbf{t}$ if there is a global state $\mathbf{s}'$ such that $(\mathbf{s}, \mathbf{t}, \mathbf{s}')$ is a step. In our example, the state (s_1, s_6) enables transition (a, g) as there exists state (s_2, s_5), so that $((s_1, s_6), (a, g), (s_2, s_5))$ is a step.

A *run* of $\mathbf{A}$ is a sequence $r = \mathbf{s}_0\mathbf{t}_1\mathbf{s}_1 \cdots \mathbf{t}_m\mathbf{s}_m$ such that $(\mathbf{s}_i, \mathbf{t}_{i+1}, \mathbf{s}_{i+1})$ is a step for every $0 \leq i \leq m-1$. Since r is completely determined by the sequence $\mathbf{t}_1 \ldots \mathbf{t}_m$, we often identify r with it. In our running example the following sequence of transitions is, e.g. a run: $r_{ex} = \langle (a,g)(c,*)(b,*)(c,*)(d,h)(f,*) \rangle$.

The *label* of a global transition $\mathbf{t} = (t_1, \ldots, t_n)$ is the tuple $\ell(\mathbf{t}) = (\ell_1(t_1), \ldots, \ell_n(t_n))$, where we extend the definition of ℓ_i by setting $\ell_i(*) = \epsilon$ for all $i \in \{1 \ldots n\}$. In our example, we have $\ell((d,h)) = (\ell_1(d), \ell_2(h)) = (\epsilon, \epsilon)$ or $\ell((c,*)) = (\ell_1(c), \ell_2(*)) = (3, \epsilon)$.

The set of *global labels* is $\mathbf{L} = (L_1 \cup \{\epsilon\}) \times \cdots \times (L_n \cup \{\epsilon\})$. The label $(\epsilon, \ldots, \epsilon)$ is the *empty (global) label.* A global transition is *invisible* if its label is empty, otherwise it is *visible.* The *observation* of a run $r = \langle \mathbf{t}_1 \ldots \mathbf{t}_m \rangle$ is the word $\ell(r) = \langle \ell(\mathbf{t}_1) \ldots \ell(\mathbf{t}_m) \rangle \in \mathbf{L}^*$, where invisible labels are omitted.

Consider the run r_{ex}, we defined in our running example. Then we have $\ell(r_{ex}) = \langle (1,2)(3,\epsilon)(3,\epsilon)(3,\epsilon)(4,\epsilon) \rangle$. Note that $|l(r)| < |r|$ as $\ell((d,h)) = (\epsilon, \epsilon)$ and with that it is invisible and omitted from the observation.

We now define the diagnosis problem we consider with respect to labelled products of transition systems. The observer records the sequence O of labels occurring in a concrete run. When an alarm occurrs we diagnose it using the observation:

Definition 16 (Diagnosis problem)**.** *A run r of $\mathbf{A}$ is an* explanation *of observation $O \in \mathbf{L}^*$ iff $\ell(r) = O$. The* diagnosis problem *is, given $\mathbf{A}$ and O, to compute the set of explanations of O.*

For our running example, assume observation $O_{ex} = \ell(r_{ex}) = \langle (1,2)(3,\epsilon)(3,\epsilon)(3,\epsilon)(4,\epsilon) \rangle$. Obviously, by definition, r_{ex} is an explanation for O_{ex}. Moreover, $r_{ex2} = \langle (a,g)(c,*)(b,*)(e,*)(f,*) \rangle$ is also an explanation for O_{ex}, as $\ell(r_{ex2}) = O_{ex}$ holds.

Since O may have infinitely many explanations, the problem is usually restricted to computing some suitable subset of explanations, which we will discuss later. Beside the general problem, we consider two interesting special cases:

- A labelling function is *perfect*, if the label of every global transition $\mathbf{t} = (t_1 \dots t_n)$ is $\ell(\mathbf{t}) = \langle \ell_1(t_1), \dots, \ell_n(t_n) \rangle$ with $\ell_i(t_i) = t_i$ for all $t_i \neq *$ and $i \in \{1 \dots n\}$. In this case the observer gets perfect information: it knows that an step occurred, and knows exactly which transitions of which transition systems where involved in the step.

- A labelling function is *participant perfect*, if the label of every global transition $\mathbf{t} = (t_1 \dots t_n)$ is $\ell(\mathbf{t}) = \langle \ell_1(t_1), \dots, \ell_n(t_n) \rangle$ with $\ell_i(t_i) \neq \epsilon$ for all $t_i \neq *$ and $i \in \{1 \dots n\}$. This models the case that the observer always knows which transition systems participate in a step, but not necessarily which transitions occurred in it.

Participant perfectioness is a necessary condition for perfectioness. Notice that, while many labelling functions used in practice are participant perfect, this is not always the case. In many models, sensors are modelled by transition systems without ϵ-transitions, while normal transition systems are modelled with *only* ϵ-transitions. The global label does then not necessarily carry perfect information about which components are interacting.

In our running example, the labelling function is not participant perfect, as the label of global transition (d, h) is $\ell((d, h)) = (\epsilon, \epsilon)$; both transition systems contribute to this global transition, but the empty label hides their contribution and one cannot tell from the observation, which transition systems participate in it. Changing this labelling function so that, e.g. $\ell((d, h)) = (1, 1)$ makes this labelling function participant perfect.

4.2 Products as Petri Nets

Products of labelled transition systems can be described as safe nets. Assume synchronized transition system $\mathbf{A} = \langle \mathcal{A}_1, \dots, \mathcal{A}_n, \mathbf{T} \rangle$. We define labelled net $N = (P, T', F, M_0, \ell')$ with ℓ' the labelling function as follows. Let f be a bijection between elements of the N and $\mathbf{A}$.

- For each state s in each transition system $\mathbf{A}$ we have a place $f(s) \in P$. Nothing else is in P

- For each global transition $\mathbf{t} \in \mathbf{T}$ there exists a transition $f(\mathbf{t}) \in T'$. Nothing else is in T'.

- For each global transition ($\mathbf{t} = \langle t_1, \ldots, t_n \rangle) \in \mathbf{T}$ and each $t_i \in \langle t_1, \cdots, t_n \rangle$ with $t_i \neq *$, :
  - we have $\langle f(\alpha(t_i)), f(\mathbf{t}) \rangle \in F$ and
  - we have$\langle f(\mathbf{t}), f(\beta(t_i)) \rangle \in F$.

  Nothing else is in F.
- For each is_i in the tuple $\mathbf{is} = \langle is_1 \ldots is_n \rangle$, $f(is_i) \in M_0$ holds. Nothing else is in M_0.
- We define labels of global transitions and respective net transition to be the same, i.e. $\ell'(f(\mathbf{t})) = \ell(\mathbf{t})$ for each global transition $\mathbf{t}$. For the reason of simplicity, and abusing language, from this moment on we also denote ℓ' by ℓ, which should cause no confusion.

We now prove that N mimics the behaviour of $\mathbf{A}$ and vice-versa by using the well-known notion of bisimulation [41], [39] adapted to our case.

Definition 17. *(Bisimulation) Let $\mathbf{S}$ be the set of all global states $\mathbf{s}$ in $\mathbf{A}$ and $\mathscr{P}(P)$ be the set of all markings of N. Let $R \subseteq \mathbf{S} \times \mathscr{P}(P)$ be some relation. We say that R is a bisimulation iff whenever $\mathbf{s} \in \mathbf{S}$, $m \in \mathscr{P}(P)$ and sRm:*

- *If there is a step $\langle \mathbf{s}, \mathbf{t}, \mathbf{s}' \rangle$, then $m \xrightarrow{t'} m'$ with $\mathbf{s}'Rm'$ for some net transition t'.*
- *If $m \xrightarrow{t'} m'$, then there is a step $\langle \mathbf{s}, \mathbf{t}, \mathbf{s}' \rangle$ with $\mathbf{s}'Rm'$ for some transition $\mathbf{t}$.*

Lemma 1. *(Bisimulation) The relation $R = \{(\mathbf{s}, m) \mid (\mathbf{s} = \langle s_1 \ldots s_n \rangle) \in S \wedge m = \{f(s_1), \ldots, f(s_n)\}\}$ is a bisimulation.*

Proof. Assume step $\langle \mathbf{s}, \mathbf{t}, \mathbf{s}' \rangle$. Transition $f(\mathbf{t})$ is enabled if all pre-places ${}^\bullet f(\mathbf{t})$ are marked. By definition of flow relation F we have ${}^\bullet f(\mathbf{t}) = \{f(\alpha_i(t_i)) \mid (t_i \neq *) \ \wedge \ i \in \{1, \ldots, n\}\}$ .This is, by definition of a step, equal to $\{f(s_i) \mid (t_i \neq *) \ \wedge \ i \in \{1, \ldots, n\}\}$. As $\mathbf{s}Rm$ holds we have that $m = \{f(s_1), \ldots, f(s_n)\}$ and with that ${}^\bullet f(t) \subseteq m$, so $f(\mathbf{t})$ is enabled. If we fire $f(\mathbf{t})$ we get marking $m' = (m \setminus {}^\bullet f(t)) \cup f(t)^\bullet$. We have $(m \setminus {}^\bullet f(t_i)) = \{f(s_i) \mid (t_i = *) \ \wedge \ i \in \{1, \ldots, n\}\}$ which is equal to $\{f(s_i') \mid (t_i = *) \ \wedge \ i \in \{1, \ldots, n\}\}$ by definition of a step. Moreover $f(t)^\bullet = \{f(\beta_i(t_i)) \mid (t_i \neq *) \ \wedge \ i \in \{1, \ldots, n\}\} = \{f(s_i') \mid (t_i \neq *) \ \wedge \ i \in \{1, \ldots, n\}\}$. Together (as the two sets are disjoint), we have $m' = \{f(s_i') \mid i \in \{1, \dots, n\}\}$ and with that $\mathbf{s}'Rm'$ holds.

Now, assume step $m \xrightarrow{t'} m'$. Transition t' exists because there is some global transition $\mathbf{t} = \langle t_1, \dots, t_n \rangle$ so that $f(\mathbf{t}) = t'$. With that, ${}^\bullet t' = \{f(\alpha_i(t_i)) \mid (t_i \neq *) \wedge i \in \{1, \dots, n\}\}$ and $t'^\bullet = \{f(\beta_i(t_i)) \mid (t_i \neq *) \wedge i \in \{1, \dots, n\}\}$. Further we have marking $m' = (m \setminus {}^\bullet t') \cup t'^\bullet$. With that we have $m' = \{f(s_i) \mid (t_i = *) \wedge i \in \{1, \dots, n\}\} \bigcup \{f(\beta(t_i)) \mid (t_i \neq *) \wedge i \in \{1, \dots, n\}\} = \{f(s_i') \mid (t_i = *) \wedge i \in \{1, \dots, n\}\} \bigcup \{f(s_i') \mid (t_i \neq *) \wedge i \in \{1, \dots, n\}\} = \{f(s_i') \mid i \in \{1, \dots, n\}\}$. Alltogether, there exists step $\langle \mathbf{s}, \mathbf{t}, \mathbf{s}' \rangle$, with $\mathbf{s}'Rm'$. $\square$

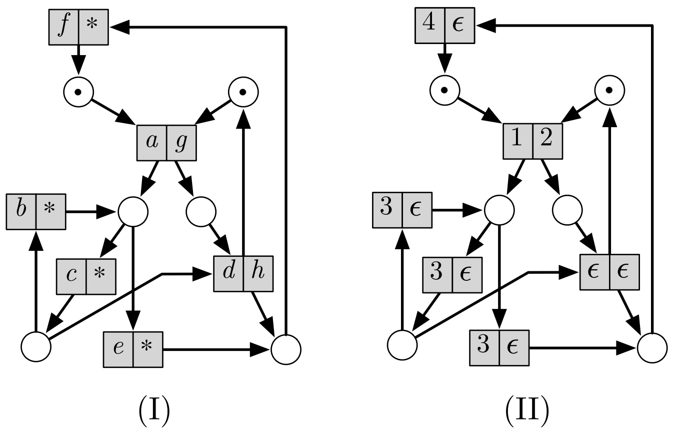


Figure 4.4: Synchronized product as Petri net, Left: with transition names, Right: with transition labels

To continue our running example consider Figure 4.4. The figure shows the synchronized product $\mathbf{A}$ of transition systems $\mathcal{A}_1$ and $\mathcal{A}_2$ with synchronization constraint $\mathbf{T}$ in its net representation.

Subfigure I shows then the names of the transitions, while Subfigure II shows their labels according to labelling function ℓ.

The diagnosis problem may be solved by constructing an automaton from A, containing global states and global transitions as automaton states and automaton transitions, respectively. The construction is as follows. Add the initial global state to the automaton, then iterate: for each added state $\mathbf{s}$ and step $\mathbf{sts}'$, add transition $\mathbf{s} \xrightarrow{\mathbf{t}} \mathbf{s}'$ to the automaton (add $\mathbf{s}'$ if it does not exist). Then, replace transitions with the respective global transition labels and keep a reference to the global transition, for reconstructing runs later. Empty labels are replaced by ϵ-transitions. The result is a non-deterministic automaton with ϵ-transitions. Construct the product of this automaton with the observation or construct the reachability graph and find the paths equal to the observation to get the desired runs. This solution is conceptually simple, but very inefficient for synchronized products with a high degree of concurrency. See Figure 4.5 for the non-deterministic automaton constructed from our running example. It represents the transition name (global transition label), with the reference to the original global transition. $(4, \epsilon)/(f, *)$, e.g. stands for transition $(4, \epsilon)$, referencing global transition $(f, *)$.

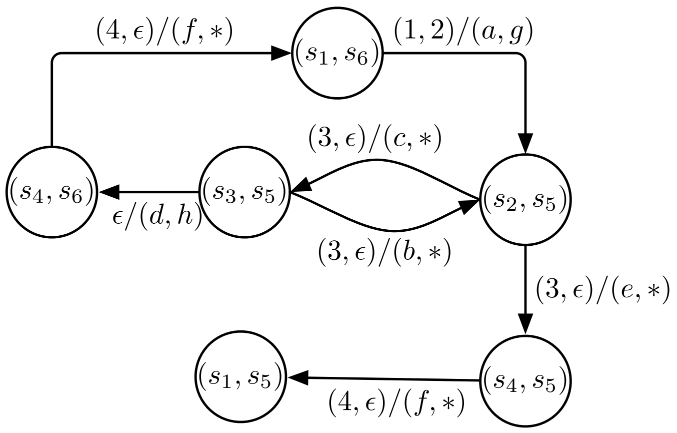


Figure 4.5: Nondeterministic automaton

In concurrent systems, it has been shown that partial order representations are very effective for addressing the state explosion problem. Following [3], we utilize the unfolding technique for representing system behaviours in a partial order compressed way, and, more importantly, we use unfoldings to represent alarm explanations.

The algorithms for generating such an unfolding are subject of the next section.

4.3 Reactive and Proactive Unfolding Algorithm

In this section, we present our algorithms for solving the diagnosis problem as defined with Definition 16. Synchronized products can be represented as Petri nets and so any set of (maximal) runs can be represented as unfolding prefix (see Section 3.2.3). As being a very effective (partial order compressed) representation for runs, we utilize prefixes as representation for sets of observation explanations. Loosely speaking, the problem we algorithmically solve in this Section is the following. Given a labelled synchronized product of transitions systems together with an observation (sequence of labels), compute all explanations for this observation, represented as unfolding prefix. For short, we call this prefix the *explain prefix*.

Basically, we distinguish between *reactive* and *proactive* algorithms for solving this problem. In the reactive case, we construct the explain prefix by iteratively adding elements to it while in the proactive case we actively speculate about the systems future while the observation sequence is not yet complete, which leads to computing behaviours that might get revoked when the observation sequence grows. These behaviours then have to be removed from the prefix in order to construct the explain prefix. Note that *reactive* and *proactive* does not correspond to offline and online. In fact, the reactive approach can be implemented both ways, offline and online; in the offline case we generate the explain prefix when the observation is complete whereas in the online case, while the system is running, we are iteratively given the growing partial observation that is a subset of the final observation and construct the explain prefix with the invariant that the unfolding prefix is at each time a prefix of the explain prefix. The proactive case, however, only makes sense when implemented online. The invariant, that the unfolding prefix is at each time a prefix of the explain prefix is dropped, however, in the end it has to hold that the constructed unfolding prefix and the explain prefix are equal.

Figure 4.6 compares the two different approaches. In this example we assume that the observer records the sequence $O = \langle\langle 1, 2\rangle\langle 2, \epsilon\rangle, \langle 2, \epsilon\rangle\rangle$. We want to diagnose that sequence, i.e. construct the respective explain prefix. On the left the construction using the offline reactive approach is sketched. We construct the explain prefix after the last label of the sequence has been

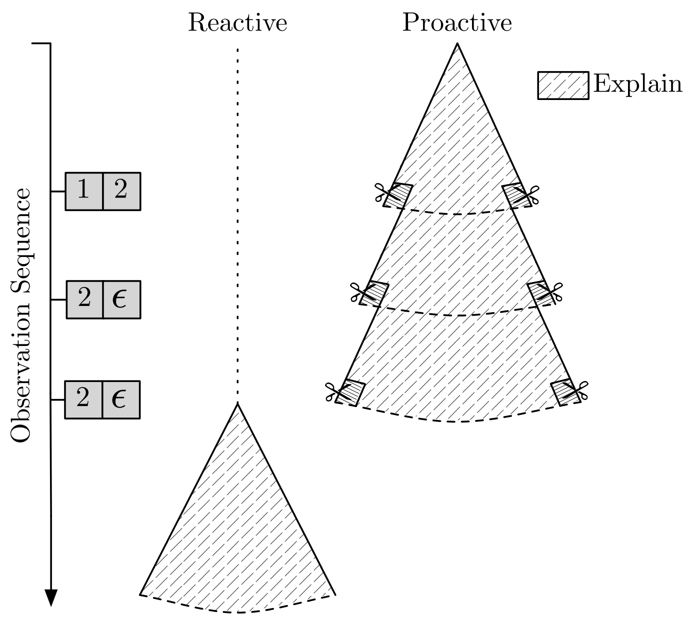


Figure 4.6: Comparison of reactive on proactive diagnosis

observed.

In contrast to that, on the right, the proactive diagnosis is shown. Even if no label has yet been observed we speculate about the future of the system. When observing a label (e.g. $\langle 1, 2\rangle$), parts of the unfolding might get obsolete and we have to remove them. This is illustrated with scissors in the Figure. We iterate this procedure, and ideally, with the last observed label we just have to apply this "garbage collection" for the last time to obtain the explain prefix.

In contrast to the related work presented in Section 3.3 we construct the explanation (unfolding) without prior synchronizing the net with the observation sequence, as the complete sequence is not at our disposal from the beginning on, but it is received iteratively.

Our approach is based on the net unfolding algorithm presented in Section 3.2.3. Therefore, we recall and briefly summarize Algorithm 6 from this section. This algorithm is a worklist algorithm. A worklist of possible extensions is maintained. Such a possible extension is an event that has not yet been added to the prefix, but all required pre-conditions are already contained and these conditions are pairwise in co-relation. We iteratively remove events from the worklist, adding them to the prefix. Whenever an event is added, it is added together with its post-conditions. It is then checked if the event is a cut-off event, i.e. an event that is not expanded further. If it is not, it is checked if new possible extensions arise from the addition of the respective new post-conditions to the prefix. Those possible extensions are added to the worklist. Initially, the prefix contains conditions that are copies of the initial marking of the net and the worklist contains all possible extensions of the those conditions.

We identify two core components of the algorithm: (1) the generation of possible extensions $\mathbf{pe}(U_p)$ for current prefix U_p and (2) checking for cut-off events. In the traditional algorithm, cut-offs are used to generate a finite complete prefix containing all reachable markings. If this check (2) is removed we have an unfolding algorithm computing the unique (possibly infinite) prefix. This is the starting point for our algorithm. We implement (1) and (2) to fit our needs for generating the explanation prefix. This prefix contains all maximal runs that "generate" the observation O. At first, we address the problem of making this notion precise.

Definition 18 (Observation and Explanation)**.** *Let $ob(\mathcal{C}) = \{\ell(r) \mid r \text{ is a realization of } \mathcal{C}\}$ be the set of observations of a configuration $\mathcal{C}$ of the unfolding. We say that $\mathcal{C}$ is an* explanation *of O if $O \in ob(\mathcal{C})$.*

Ideally, we would like a finite prefix containing all explanations of O. However, if the reachability graph contains loops on invisible transitions, the set of explanations can be infinite and no finite prefix exists. So we have to lower our aim. For that, we introduce the concept of verbose and succinct explanations.

Definition 19 (Verbose and Succinct)**.** *An explanation $\mathcal{C}$ of O is* verbose *if it contains events e, e' such that $[e] \subset [e'] \subseteq \mathcal{C}$, $ob([e]) = ob([e'])$, and $Mark([e]) = Mark([e'])$. If $\mathcal{C}$ is not verbose, then it is* succinct*.*

Given a realization $rr'r''$ of verbose explanation $\mathcal{C}$ of O such that r and rr' are realizations of $[e]$ and $[e']$, with $\ell(rr'r'') = O$; respectively the sequence rr'' is a run and also satisfies $\ell(rr'') = O$.

A verbose explanation contains two causally dependent events so that their local configurations produce the same local marking and have the same observation.

An observation O may have infinitely many explanations, but only finitely many succinct explanations. See for that the example in Figure 4.7 where

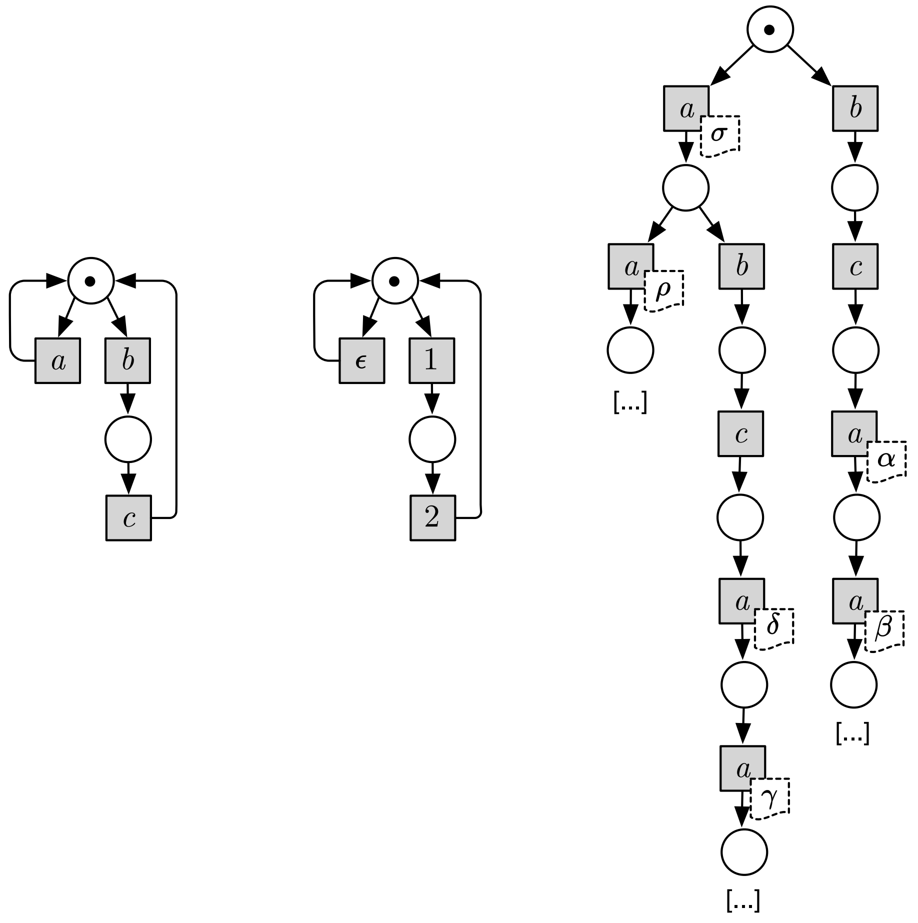


Figure 4.7: Left: Petri net representing synchronized product, Middle: Petri net from left with labels, Right: unfolding

three nets are shown. The most left one is a Petri net, representing a synchronized product of transition systems with global transitions a, b and c. The net in the middle is its labelled representation. We have following labelling function ℓ: $\ell(a) = \epsilon$, $\ell(b) = 1$ and $\ell(c) = 2$. Assume observation sequence $O = \langle 1, 2 \rangle$. Then, the unfolding prefix on the right of the figure is a prefix of the infinite explain prefix. Events carry the name of the respective transition. To be able to refer to certain events, some of them carry an additional Greek label. The most right "path" in the unfolding ($\langle b, c, a, a \rangle$) is an explanation, as its observation is O. However, as $[\alpha] \subset [\beta]$, $ob([\alpha]) = ob([\beta])$ and $Mark([\alpha]) = Mark([\beta])$, it is a verbose one. The same holds for pairs of events (σ, ρ), (δ, γ) and infinitely more. Removing the successors of σ, δ and α, we have a finite prefix containing all succinct explanations.

In general, the argument for O having only finitely many succinct explanations is as follows. Each chain of invisible events in a succinct explanation contains at most as many events as the amount of reachable markings of the net. Assume there exists a chain of invisible events containing more markings. Then, due to the pigeon principle, there exist two events e, e' in this chain with $Mark([e]) = Mark([e'])$. As this chain only contains invisible events, we have $ob([e]) = ob([e'])$. But then the explanation is not succinct, which is a contradiction. With that, a prefix containing all succinct explanations is finite.

On the one hand, we want to generate a prefix containing all succinct explanations. This is formulated using O-completeness:

Definition 20 (O-completeness)**.** *An unfolding prefix is* O-complete *if it contains all succinct explanations.*

On the other hand, intuitively, we also would like that every configuration of the branching process is either an explanation of O or a subset of it:

Definition 21 (O-compatibility)**.** *A configuration $\mathcal{C}$ is* O-compatible *if it can be extended to an explanation, i.e. if some configuration $\mathcal{C}' \supseteq \mathcal{C}$ is an explanation of O. An unfolding prefix is O-compatible if, for each event e, the local configuration $\lceil e \rceil$ is O-compatible.*

Ideally, given a possible extension $e \in \mathbf{pe}(U_p)$ in the procedure above, we would like to add the check if the local configuration $[e]$ is O-compatible with O and extend U_p with e if and only if the answer is affirmative. However, this problem is PSPACE-hard, as shown in the following lemma.

Lemma 2. *Deciding whether a configuration C is O-compatible to observation O is PSPACE-hard.*

Proof. We reduce the reachability problem in 1-safe Petri nets to this problem. The reachability problem (PSPACE-complete, see [4]) is, given a net $N = (P, T, F, M_0)$ with initial marking M_0 and target marking M_f, to decide if there exists a run $\sigma = t_1 \dots t_n$, i.e. firing sequence $M_0 \xrightarrow{t_1} M_1 \cdots \xrightarrow{t_n} M_f$. Let ${M_f}^m$ and ${M_f}^{!m}$ be the sets of places which are marked and not marked, respectively, by M_f. We create the net $N' = (P' = P, T' = T, F' = \varnothing, M_0' = M_0)$. Further, let f be a bijection on places and transitions between N' and N.

For once, we add a transition $t_{reached}$ to T'. For each place $p' = f(p) \in P'$ we add places $b_{p'}, a_{p'}, s_{p'}$ to P', transitions $enter_{p'}$, $leave_{p'}$ to T' and $(s_{p'}, t_{reached}), (b_{p'}, enter_{p'}), (enter_{p'}, p'), (p', leave_{p'})$ and $(leave_{p'}, a_{p'})$ to F. If $p' \in {M_f}^{!m}$ we add $s_{p'}$ to M'_0. Further, for each $t \in {}^\bullet p$ we add pair $(f(t), p'_b)$ to F' and for each $t \in p^\bullet$ we add pair $(p'_a, f(t))$ to F'. For each transition $p' = f(p) \in {M_f}^{!m}$, add $(s_{p'}, enter_{p'})$ and $(leave_{p'}, s_{p'})$ to F. For each transition $p' = f(p) \in {M_f}^{m}$, add $(s_{p'}, leave_{p'})$ and $(enter_{p'}, s_{p'})$ to F.

Figure 4.8 visualises this construction for two places $p'_1 = f(p_1)$ and $p'_2 = f(p_2)$, where p_1 is marked by M_f and p_2 not.

Observe that transition $t_{reached}$ can be enabled iff M_f is reachable. We set this transition $t_{reached}$ to be the only visible transition with $\ell(t_{reached}) =$ *reached* and we set observation $O = \langle reached \rangle$. The question if the empty configuration $\mathcal{C}$ is O-compatible with O in the unfolding of N' is then the question if M_f is reachable in N. With that, O-compatibility is PSPACE-hard.

□

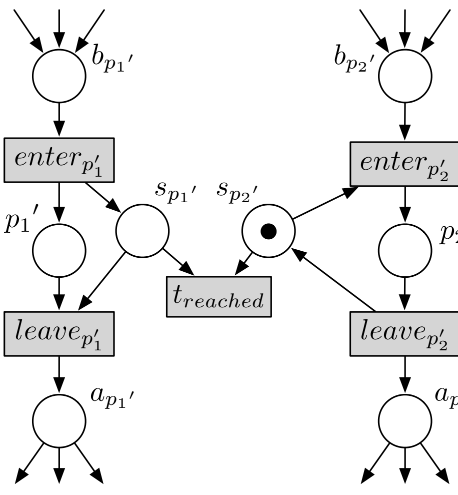


Figure 4.8: Illustration for reducing the reachability problem to the O-compatibility problem

So again we lower our aim, for which we need some basic notions of trace theory [11].

Definition 22. *(Dependent/Independent transitions and labels) Two global* *transitions* $\mathbf{t} = \langle t_1, \ldots, t_n \rangle$ *and* $\mathbf{t}' = \langle t'_1, \ldots, t'_n \rangle$ *are* independent *if* $t_i \neq *$ *implies* $t'_i = *$ *for every* $i \in \{1, \ldots, n\}$*, i.e. no transition system contributes to both global transitions. If not they are* dependent*.*

Similarly, two global labels $\ell = \langle \ell_1, \ldots, \ell_n \rangle$ *and* $\ell' = \langle \ell'_1, \ldots, \ell'_n \rangle$ *are* independent *if* $\ell_i \neq \epsilon$ *implies* $\ell'_i = \epsilon$ *for every* $i \in \{1, \ldots, n\}$*. If not they are* dependent*.*

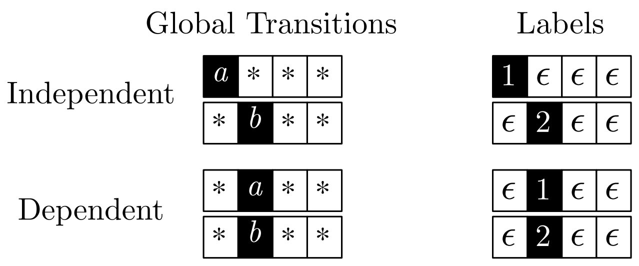


Figure 4.9: Dependent and independent global transitions and labels

Figure 4.9 shows an example for that. Four cells in a row illustrate a global transition (left) or a global label (right), synchronizing transitions or labels of four transition systems. For illustration purposes, we draw the background of cells, with labels that are not epsilon and transitions that are not idling, in black. Loosely speaking two labels or two global transitions are independent if there are no two black labels in one column.

It is well-known that any two realizations of the same configuration are meaning-equivalent. In trace theory we have a similar notion; equivalence. We define equivalence between global transition sequences $\sigma \equiv \sigma'$ and observations $O \equiv O'$ the same way:

Definition 23 (Equivalence)**.** *For two sequences* $\sigma, \sigma' \in \mathbf{T}^*$ *or two observations* O *and* $O' \in \mathbf{L}^*$ *we write* $\sigma \equiv_1 \sigma'$ *or* $O \equiv_1 O'$*, respectively, if* σ' *is obtained from* σ *by swapping two independent global transitions or* O *from* O' *by swapping two independent labels, respectively. More precisely,* $\sigma \equiv_1 \sigma'$ *if* $\sigma = \mathbf{t_1} \ldots \mathbf{t_i} \mathbf{t_{i+1}} \ldots \mathbf{t_k}$ *and* $\sigma' = \mathbf{t_1} \ldots \mathbf{t_{i+1}} \mathbf{t_i} \ldots \mathbf{t_k}$ *or* $O \equiv_1 O'$ *if* $O = \ell_1 \ldots \ell_i \ell_{i+1} \ldots \ell_k$ *and* $O' = \ell_1 \ldots \ell_{i+1} \ell_i \ldots \ell_k$ *for some index i such that* $\mathbf{t_i}$ *and* $\mathbf{t_{i+i}}$ *or* ℓ_i *and* ℓ_{i+1} *are independent, respectively.*

We denote the transitive closure of $\equiv_1$ *by* $\equiv$*, and say that* O *and* O' *are* equivalent *if* $O \equiv O'$ *(the same for* σ *and* σ'*).*

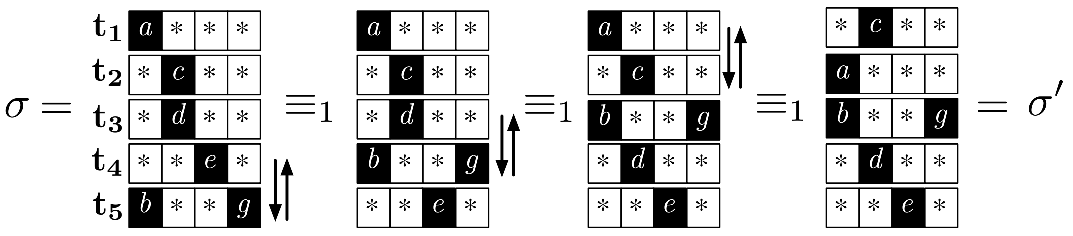


Figure 4.10: Equivalence of sequences

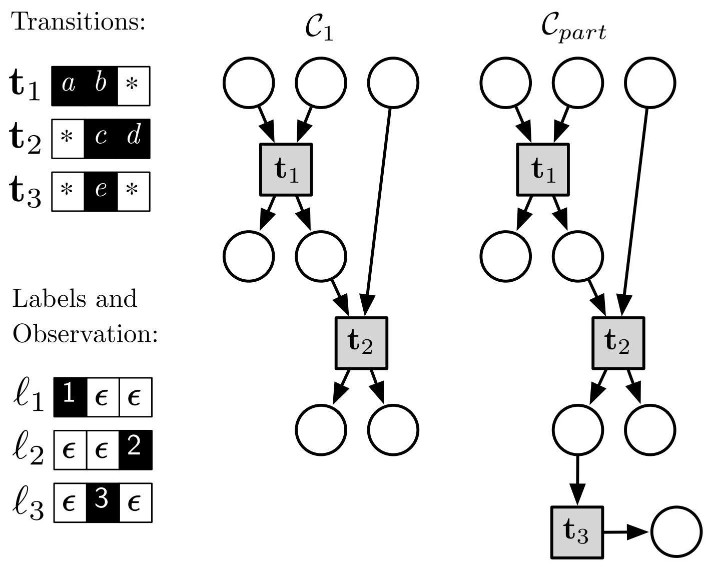
Figure 4.10 shows an example for that. There, a sequence σ of global transitions $\langle \mathbf{t_1}, \ldots, \mathbf{t_5} \rangle$ is shown on the left, which is equivalent to the sequence σ' on the right ($\sigma \equiv \sigma'$). We get σ' from σ by iteratively swapping independent global transitions.

Observe that, since local transitions can be labelled with ϵ, if two global transitions are independent then their labels are independent, but the converse only holds if the labelling function is participant perfect. This extends to observations: if $\sigma \equiv \sigma'$, then $\ell(\sigma) \equiv \ell(\sigma')$, but the converse only holds for participant perfect labellings.

With that we define O-soundness, i.e. an approximation for O-completeness.

Definition 24 (O-soundness)**.** *A configuration $\mathcal{C}$ is O-*sound*, if for every $O_1 \in ob(\mathcal{C})$ there is $O_2 \in \mathbf{L}^*$ such that $O_1O_2 \equiv O$.*

Figure 4.11 compares O-compatibility and O-soundness. In this example we have three global transitions $\mathbf{t_1}, \mathbf{t_2}$ and $\mathbf{t_3}$ with respective global labels ℓ_1, ℓ_2 and ℓ_3. Let $\mathcal{N}$ be the net representation of some synchronized product so that $\mathcal{C}_1, \mathcal{C}_{part}$ and $\mathcal{C}_2$ are configurations of its unfolding. Events are labelled with the respective original transition. Let $O = \langle \ell_1, \ell_2, \ell_3 \rangle$ be the recorded observation. Note that the observation of the realization of $\mathcal{C}_{part}$ is also O, so it is O-compatible. Moreover, $\mathcal{C}_1$ is O-compatible as there exists a set of events E so that $\mathcal{C}_1 \overset{E}{\leadsto}$, i.e. $\mathcal{C}_1$ extended with E is a configuration, and there exists a realization of this configuration with observation O. Here, E solely contains the event labelled with $\mathbf{t_3}$. Assume that there exists no set of events E' so that $\mathcal{C}_2 \overset{E'}{\leadsto}$ has observation O. Therefore, $\mathcal{C}_2$ is not O-compatible. However, $\mathcal{C}_2$ is O-sound as its observation is $\langle \ell_2, \ell_1 \rangle$, both labels are independent and with choosing $O_1 = \langle \ell_2, \ell_1 \rangle$ and $O_2 = \langle \ell_3 \rangle$, we have $O_1O_2 = \langle \ell_2, \ell_1, \ell_3 \rangle \equiv \langle \ell_1, \ell_2, \ell_3 \rangle = O$. We now show that O-soundness is a



Transitions:
$\mathbf{t}_1$ a b $*$
$\mathbf{t}_2$ $*$ c d
$\mathbf{t}_3$ $*$ e $*$
Labels and Observation:
ℓ_1 1 ϵ ϵ
ℓ_2 ϵ ϵ 2
ℓ_3 ϵ 3 ϵ
$\mathcal{C}_1$
$\mathcal{C}_{part}$
$\mathcal{C}_2$
$\mathbf{t}_1$
$\mathbf{t}_2$
$\mathbf{t}_3$

Figure 4.11: Example for comparing O-soundness with O-compatibility

necessary condition for compatibility, i.e. if a configuration is not O-sound it is also not O-compatible. The other direction must not necessarily hold, as already shown with the counterexample from Figure 4.11.

Proposition 1. *If $\mathcal{C}$ is compatible with O, then $\mathcal{C}$ is O-sound.*

Proof. We need some preliminaries. We extend the notions of independence and equivalence to sequences of global transitions.

For the proof, assume $\mathcal{C}$ is compatible with O, and let $O_1 \in ob(\mathcal{C})$. By the definition of compatibility, there is a Configuration C' with $\mathcal{C}' \supseteq \mathcal{C}$ such that $O \in ob(\mathcal{C}')$. Let r_1 be a realization of $\mathcal{C}$ such that $\ell(r_1) = O_1$. Since $\mathcal{C}' \supseteq \mathcal{C}$, there is a realization $r' = r_1 r_2$ of $\mathcal{C}'$, and so O_1 is a prefix of the observation $O' = \ell(r_1)\ell(r_2)$ of $\mathcal{C}'$. It remains to prove $O \equiv O'$.

Let r be a realization of $\mathcal{C}'$ such that $\ell(r) = O$. Then we have $r \equiv r'$. Since $\ell(t) \neq \epsilon$ implies $t \neq *$, if two global transitions $\mathbf{t}, \mathbf{t}'$ are independent then $\ell(\mathbf{t}), \ell(\mathbf{t}')$ are also independent, and so $r \equiv r'$ implies $O = \ell(r) \equiv \ell(r') = O'$. So $\mathcal{C}$ is O-sound. □

We say that a branching process is *O-sound* if all its configurations are O-sound. We provide a diagnosis algorithm that constructs an O-sound and O-complete branching process. The algorithm itself is again a simple modification of the standard algorithm for the construction of complete prefixes. First we define O-cut-offs. We denote by $\mathbf{pe}_{\mathbf{sound}}(U_p)$ the set of events $e \in \mathbf{pe}(U_p)$ such that $\lceil e \rceil$ is O-sound.

Definition 25 (O-cut-off)**.** *An event $e \in \mathbf{pe}_{\mathbf{sound}}(U_p)$ is an* O-cut-off *if there is an event $e' < e$ such that $Mark(e) = Mark(e')$ and $ob([e]) = ob(e')$.*

In the beginning of this section we already identified the two core components of Algorithm 6: (1) computing possible extensions and (2) computing cut-offs. The *diagnosis procedure* is the result of substituting $\mathbf{pe}(U_p)$ in (1) with $\mathbf{pe}_{\mathbf{sound}}(U_p)$ and the cut-off procedure in (2) with observation cut-offs. In the rest of this section we prove the following theorem:

Theorem 1. *The diagnosis procedure terminates with an O-sound and O-complete unfolding prefix `Explain`.*

Let us first prove termination. The diagnosis procedure terminates iff `Explain` is finite (as each iteration of the while loop adds new elements to `Explain`). Let $length(ob([\mathcal{C}]))$ be the common length of all the observations of $ob([\mathcal{C}])$. We have:

Lemma 3. *For every event e of `Explain` $length(ob([e])) \leq |O|$.*

Proof. If there is an observation $O_l \in ob(e)$ such that $|O_l| > |O|$, then there exists no $O_2 \in \mathbf{L}^*$ such that $O_l O_2 \equiv O$, and so $e \notin Ext(\mathcal{N}, CO, O)$ for every $\mathcal{N}$ and CO. So e is never added to `Explain`. □

A sequence $\pi = e_1 \dots e_n$ of events is *a path* of e iff ${}^{\bullet}({}^{\bullet}e_1) = \varnothing$, $e_n = e$, and $e_{k-1} \in {}^{\bullet}({}^{\bullet}e_k)$ for any pair of successive events e_{k-1}, e_k. Let $\mathcal{H}(e)$ be the maximal length of the paths of e. An event e_k of a path of e is *observable* if $ob([e_k]) \neq ob([e_{k-1}])$.

Lemma 4. *The paths of the events of `Explain` contain at most $|O|$ observable events.*

Proof. Observe that e_k is observable iff $length(ob([e_{k-1}])) > length(ob([e_k]))$. As $length(ob([e]))$ is bounded by $|O|$ for any event $e \in$ `Explain`, and the observation length grows with each observable event, we have at most $|O|$ observable events. □

Lemma 5. *For every event e of `Explain` we have $\mathcal{H}(e) \leq (|O| \cdot \mathbf{R}) + 1$, where $\mathbf{R}$ is the number of reachable global states of the product.*

Proof. Let $\pi = e_1 \ldots e_m$ be a path of e. Assume $m > |O| \cdot \mathbf{R} + 1$. Since $\mathit{length}([e_i]) \leq |O|$ for every $1 \leq i \leq m$, by the pigeonhole principle there exist two events e_i and e_j such that $i < j < m$, $Mark(e_i) = Mark(e_j)$, and $ob([e_i]) = ob([e_j])$. But then e_j is a cut-off, contradicting that $e = e_m$ is an event of `Explain`. □

In summary, we have $\mathcal{H}(e) < (|O| \cdot \mathbf{R}) + 1$. It is well-known (see e.g. [15]) that in any unfolding, the sets ${}^\bullet e$ and $e^\bullet$ are finite, as well the number of events of a certain height. So `Explain` is finite and our construction procedure terminates. We continue with O-soundness and O-completeness.

Lemma 6. *`Explain` is O-sound.*

Proof. Let $\mathcal{C}$ be an arbitrary configuration of `Explain`. We prove that $ob(\mathcal{C})$ contains a prefix of O. The proof is by induction on $|\mathcal{C}|$. If $|\mathcal{C}| = 0$ then $ob(\mathcal{C}) = \epsilon$ and there is nothing to prove. If $|\mathcal{C}| > 0$, then there is a configuration $\mathcal{C}'$ and an event $e \notin \mathcal{C}'$ of `Explain` such that $\mathcal{C} = \mathcal{C}' \cup \{e\}$. Let $\mathcal{C}'' = \mathcal{C}' \cap [e]$. By induction hypothesis, there exists a prefix O' of O such that $O' \in ob(\mathcal{C}')$. Since e was added to `Explain` by the algorithm, there also exists a further prefix O^e of O such that $O^e \in ob([e])$. Moreover, we can choose O' and O^e such that $O' = O''O'_1$ and $O^e = O''O^e_1$ for some $O'' \in ob(\mathcal{C}' \cap [e])$ and some observation sequences O'_1 and O^e_1. We prove that $O''O'_1O^e_1$ is an observation of $\mathcal{C}$ and a prefix of O. For the first part, we observe that any two events $e_1 \in \mathcal{C}' \setminus [e]$ and $e_2 \in [e] \setminus \mathcal{C}'$ are concurrent. Therefore, there exists a realization r of $\mathcal{C}$ in which all events of $[e] \setminus \mathcal{C}'$ appear after all events of $\mathcal{C}' \setminus [e]$. Since $\ell(r) = O''O'_1O^e_1$, we are done. For the second part, we observe that the labels of two concurrent events are either both empty, or distinct. Indeed, if two events e_1, e_2 are concurrent, then the sets of participants and their global transitions are disjoint. So, if their labels ℓ_1, ℓ_2 are nonempty, then there are indices $i \neq j$ such that $\ell_{1i} \neq \varnothing = \ell_{2i}$ and $\ell_{1j} = \varnothing \neq \ell_{2j}$. □

Lemma 7. *`Explain` is O-complete.*

Proof. Assume `Explain` is not O-complete. Then there exists a succinct explanation $\mathcal{C} \notin$ `Explain`, so there exists a cut-off event $e_c \in \mathcal{C}$ or there exists an event $e_p \in \mathcal{C}$ such that $e_p \notin Ext(\texttt{Explain}, CO, O)$ (or both). If $\mathcal{C}$ contains a cut-off event e_c, then there exists an event e'_c such that

$[e'_c] \subset [e_c] \subseteq \mathcal{C}$, $ob([e_c]) = ob([e'_c])$, and $Mark([e_c]) = Mark([e'_c])$, but then $\mathcal{C}$ is a verbose explanation. If $\mathcal{C}$ contains an event $e_p \notin Ext(\texttt{Explain}, CO, O)$, then there exists $O_1 \in ob([e_p])$ such that for all $O_2 \in \mathbf{L}^*$, $O_1 O_2 \not\equiv O$ and with that $O \notin ob(\mathcal{C})$, but then $\mathcal{C}$ is no explanation. $\square$

Implementation of the Reactive Diagnosis

The core of any procedure for constructing complete branching processes is the routine for the computation of $\mathbf{pe}(U_p)$. Similarly, the key of the diagnosis procedure is the routine for computing $\mathbf{pe}_{\mathbf{sound}}(U_p)$. Since $\mathbf{pe}_{\mathbf{sound}}(U_p) \subseteq \mathbf{pe}(U_p)$, we use existing procedures for computing $\mathbf{pe}(U_p)$ [15], and for each event $e \in \mathbf{pe}(U_p)$ we check the additional condition: whether $[e]$ is O-sound.

However, even though O-soundness is easier to check than O-compatibility, it is still a very costly procedure. In particular, it involves a check of the form $O_1 O_2 \equiv O$ for every event e, which cannot be implemented very efficiently. We provide a more efficient approach that preserves O-completeness but not necessarily O-soundness, i.e. some runs of the prefix may be spurious explanations. It is based on the following non-trivial, but well-known result [11, Chapter 1.5][1]:

Proposition 2. *Given a realization r of a configuration $\mathcal{C}$, let r_i denote the projection of r onto the global transitions in which $\mathcal{A}_i$ participates. Two runs r, r' are realizations of the same configuration $\mathcal{C}$ if and only if $r_i = r'_i$ for every $i \in \{1, \ldots, n\}$.*

Consider Figure 4.12 as example for that. It shows a configuration $\mathcal{C}$ with two realization “Run 1” and “Run 2”. Observe that both runs share the same projection to transition systems $T_1 \ldots T_3$, as shown on the right of the figure. More general, every realization of $\mathcal{C}$ shares this projection, and each run with this projection is a realization of $\mathcal{C}$.

We exploit this proposition. Our algorithm stores for every event only the projection of its labels to respective transition systems:

Definition 26. *(Footprint and Projection) Let r be some run of a labelled product and $\ell(r) = \langle \ell(r_1), \ldots, \ell(r_n) \rangle$ the respective sequence of labels. The tuple $proj(\ell(r)) = \langle \ell(r)_1, \ldots, \ell(r)_n \rangle$ where $\ell(r)_i$ denotes the projection of*

[1] In [11] the result is formulated in the terminology of Mazurkiewicz traces.

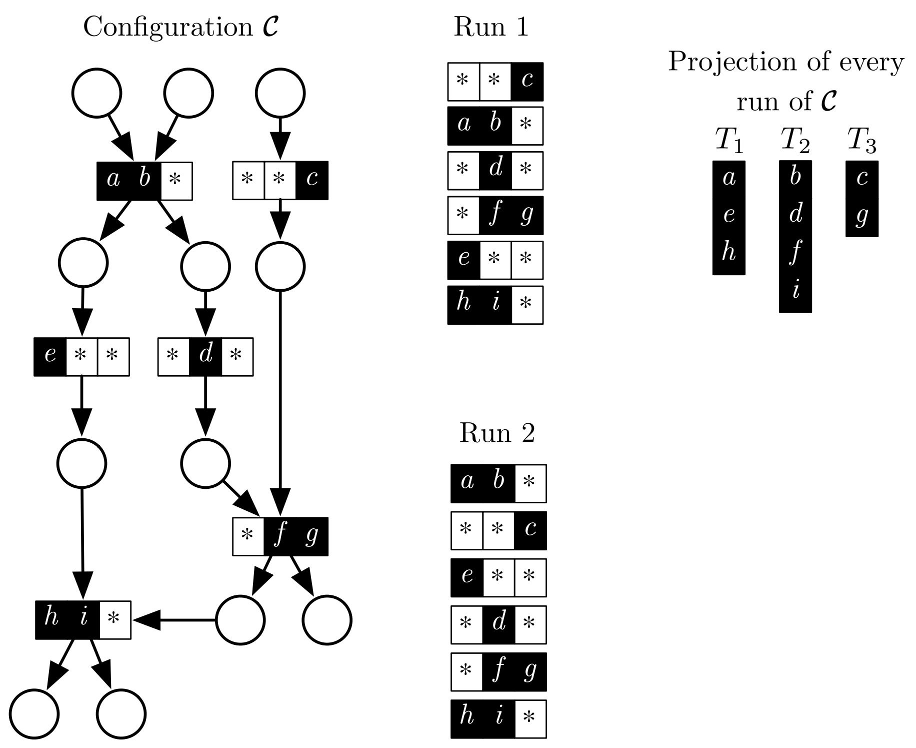
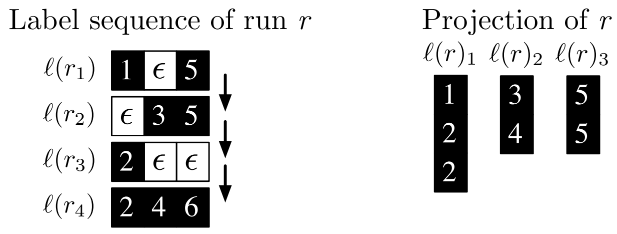

Configuration $\mathcal{C}$
Run 1
Projection of every run of $\mathcal{C}$
T_1 T_2 T_3
Run 2
a b e h i d f g c *

Figure 4.12: Example for run projections. Left: configuration, Middle: runs of configuration, Right: unique projection.

$\ell(r)$ onto the labels $\ell = (\ell_1, \ldots, \ell_n)$ such that $\ell_i \neq \epsilon$, is the projection *of $\ell(r)$. The projection proj$(\mathcal{C})$ of some configuration $\mathcal{C}$ is the unique projection proj$(\ell(r))$ of some run r of $\mathcal{C}$. We say that the footprint of observation O is its projection proj(O). Furthermore, with proj(e) for some event e, we denote the projection proj$([e])$.*

Figure 4.13 shows on the left the sequence of labels $\ell(r)$ of run r in a labelled product of 3 transition systems. We have $\ell(r) = \langle \ell(r_1), \ldots, \ell(r_4) \rangle = \langle \langle 1, \epsilon, 5 \rangle, \langle \epsilon, 3, 5 \rangle, \langle 2, \epsilon, \epsilon \rangle, \langle 2, 4, 6 \rangle \rangle$. The projection *proj*$(\ell(r))$ is shown on the right, a tuple $\langle \ell(r)_1, \ell(r)_2, \ell(r)_3 \rangle = \langle \langle 1, 2, 2 \rangle, \langle 3, 4 \rangle, \langle 5, 5 \rangle \rangle$, each element is a projection of the labels of one transition system, omitting transitions labelled with ϵ.

Proposition 2 guarantees that *proj*$([e])$ is independent of the realization. Given an event e with immediate predecessors $e_1, \ldots, e_k$, *proj*$([e])$ can be easily computed from *proj*$([e_1]), \ldots,$ *proj*$([e_k])$. Let *proj*$([e_i]) = \langle \sigma_{i1}, \ldots, \sigma_{in} \rangle$ with $i \in \{1, \ldots, k\}$ and let n be the number of participating transition



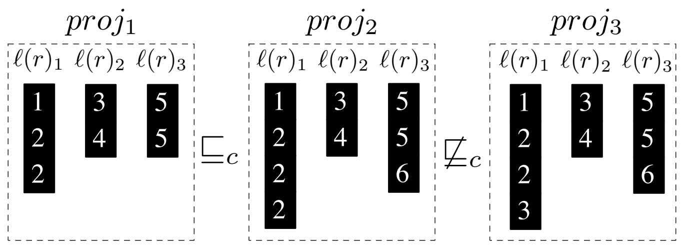
Label sequence of run r
$\ell(r_1)$ 1 ϵ 5
$\ell(r_2)$ ϵ 3 5
$\ell(r_3)$ 2 ϵ ϵ
$\ell(r_4)$ 2 4 6
Projection of r
$\ell(r)_1$ $\ell(r)_2$ $\ell(r)_3$
1 2 2
3 4
5 5

Figure 4.13: Example for a footprint. Left: label sequence, Right: projection.

systems. Since $e_1, \ldots, e_k$ are pairwise concurrent, we have $\mathit{proj}([e] \setminus \{e\}) = \mathit{proj}([e_1] \cup \cdots \cup [e_k]) = \langle \sigma'_1, \ldots, \sigma'_n \rangle$, where σ'_i is the longest of $\sigma_{i1}, \ldots, \sigma_{ik}$, and so $\mathit{proj}([e]) = (\sigma'_1 \cdot \ell_1, \ldots, \sigma'_n \cdot \ell_n)$, where ℓ_i is the label of the global transition of e, if $\mathcal{A}_i$ participates in it, and the empty word otherwise.

We say a projection is prefix of another projection if it is one component wise (Operator $\sqsubseteq_c$).

For example, Figure 4.14 shows three projections. We have $\mathit{proj}_1 \sqsubseteq_c \mathit{proj}_2$, because $\langle 1,2,2 \rangle \sqsubseteq \langle 1,2,2,4 \rangle$, $\langle 3,4 \rangle \sqsubseteq \langle 3,4 \rangle$ and $\langle 5,5 \rangle \sqsubseteq \langle 5,5,6 \rangle$. Also, we have $\mathit{proj}_2 \not\sqsubseteq_c \mathit{proj}_3$, because $\langle 1,2,2,2 \rangle \not\sqsubseteq \langle 1,2,2,3 \rangle$.



$proj_1$
$\ell(r)_1$ $\ell(r)_2$ $\ell(r)_3$
1 2 2
3 4
5 5
$\sqsubseteq_c$
$proj_2$
$\ell(r)_1$ $\ell(r)_2$ $\ell(r)_3$
1 2 2 2
3 4
5 5 6
$\not\sqsubseteq_c$
$proj_3$
$\ell(r)_1$ $\ell(r)_2$ $\ell(r)_3$
1 2 2 3
3 4
5 5 6

Figure 4.14: Example for comparing projections; proj_1 is subset of proj_2 but proj_2 is no subset of proj_3.

Our algorithm replaces the check "is $[e]$ O-sound?" by "is $\mathit{proj}([e])$, componentwise, a prefix of $\mathit{proj}(O)$?" ($\mathit{proj}([e]) \sqsubseteq_c \mathit{proj}(O)$). More precisely, if we denote by $\mathbf{pe}_{\mathbf{projection}}(U_p)$ the new possible extension function with this new check, then our implementation of the diagnosis procedure is the result of replacing $\mathbf{pe}(U_p)$ by $\mathbf{pe}_{\mathbf{projection}}(U_p)$.

The following proposition shows that the new algorithm is still O-complete: if it excludes an event from the prefix, then the old algorithm would have also excluded it.

Proposition 3. *If $proj([e])$ is not componentwise a prefix of $proj(O)$, then $[e]$ is not O-sound.*

Proof. Assume the contrary. There exists a component i such that $proj([e])_i \not\sqsubseteq proj(O)_i$ and for every $O_1 \in ob([e])$ there is a $O_2 \in \mathbf{L}^*$ such that $O_1 O_2 \equiv O$. For all such $O_1 O_2$ we have $proj(O_1 O_2)_i = proj(O_1)_i proj(O_2)_i$, as $[e]$ is a configuration. Furthermore, by definition, $proj(O_1)_i = proj(e)_i$, so $proj(O_1)_i \not\sqsubseteq proj(O)_i$ and so $proj(O_1 O_2)_i \neq proj(O)_i$, contradicting that, since $O_1 O_2 \equiv O$, all its projections must be equal. □

Termination of the diagnosis procedure is also not affected. However, the new `Explain` branching process may contain configurations that do not correspond to an explanation. Consider the product of transition systems informally described as follows, where $q \xrightarrow{\ell} q'$ denotes a transition labelled by ℓ with source q and target q'. $\mathcal{A}_1$ has transitions $t_1 = q_1 \xrightarrow{a} q_2$ and $t_2 = q_2 \xrightarrow{\epsilon} q_3$, and $\mathcal{A}_2$ has a transition, $t_2' = r_1 \xrightarrow{b} r_2$. The synchronization constraint is $\{\mathbf{u_1} = \langle t_1, * \rangle, \mathbf{u_2} = \langle t_2, t_2' \rangle\}$. The product has one single run $r = \mathbf{u_1 u_2}$ and $\ell(r) = \langle \ell(\mathbf{u_1}), \ell(\mathbf{u_2}) \rangle = \langle \langle a, \epsilon \rangle \langle \epsilon, b \rangle \rangle$. There are no explanations at all for an observation $O = \langle \langle \epsilon, b \rangle \langle a, \epsilon \rangle \rangle$. However, since the footprint of O is $proj(O) = \langle \langle a, \epsilon \rangle, \langle \epsilon, b \rangle \rangle$, and the footprint of the only event e having r as realization satisfies $proj([e]) = \langle \ell(u_1), \ell(u_2) \rangle = \langle \langle a, \epsilon \rangle, \langle \epsilon, b \rangle \rangle$, the configuration of $[e]$ belongs to `Explain` for O. The reason for this is that the labelling function is not participant perfect: the global transition $\mathbf{u_2}$ is jointly executed by $\mathcal{A}_1$ and $\mathcal{A}_2$, but its global label is $\langle \epsilon, b \rangle$, hiding the participation of $\mathcal{A}_1$. We have:

Proposition 4. *If the labelling function of the product is participant perfect, then the branching processes `Explain` computed with the help of $\mathbf{pe}_{\mathbf{projection}}(U_p)$ instead of $\mathbf{pe}(U_p)$ is O-sound and O-complete.*

Proof. It suffices to prove O-soundness for every configuration $[e]$ of `Explain`; the rest is shown as in Lemma 6. By definition, $proj([e])$ is componentwise a prefix of $proj(O)$. By [11, Chapter 1.5], the set of realizations of $proj([e])$ and $proj(O)$ according to the independence relation on labels is completely determined by $proj([e])$ and $proj(O)$; moreover, since $proj([e])$ is componentwise a prefix of $proj(O)$ there is a realization O' of $proj([e])$ that is a prefix of O. Since the labelling function is participant perfect, any two global transitions of $[e]$ are independent iff their labels are independent. Therefore, there is a realization r of $[e]$ such that $\ell(r) = O'$. $\square$

4.3.1 Analyzing the Branching Process

The diagnosis procedure yields a prefix `Explain` containing all succinct explanations of the observation O, which usually corresponds to an error or an undesirable situation. Explaining the error cause usually needs some additional reasoning on `Explain`, e.g. answering queries like "what transitions are contained in every run leading to the error?". In our implementation we use a SAT Solver to analyze `Explain`.

Algorithm 7: Branching process to CNF conversion

```
Data: Occurrence net $\mathcal{O} = \{\text{Events } E, \text{Conditions } C\}$
Result: CNF Formula (each "add" adds a clause)
for $c \in C$ do
    $e_p = \text{pre\_event}(c)$
    for $e \in \text{post\_events}(c)$ do
        for $e_1 \in \text{post\_events}(c) \setminus e$ do
            add($e \Rightarrow \neg e_1$)
        add($e \Rightarrow e_p$) (drop clause if $e_p$ is non-existent)
        add($e \Rightarrow \neg c$)
    add($\bigvee(\text{post\_events}(c)) \vee c \vee \neg e_p$) (drop literal $\neg e_p$ if $e_p$ is non-existent)
    add($c \Rightarrow e_p$) (drop clause if $e_p$ is non-existent)
```

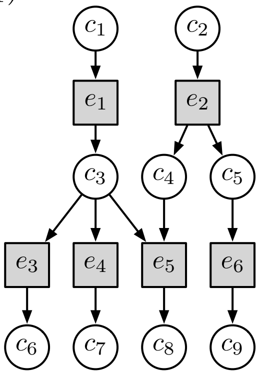
It is a well-known fact that, given some occurrence net $\mathcal{O}$, one can construct a boolean formula ϕ in linear time (see Algorithm 7 for construction), whose set of models is isomorphic to the set of configurations of $\mathcal{O}$. More precisely, ϕ contains a variable x_e for each event e, and $\mathcal{C}$ is a configuration of `Explain` iff there exists an assignment that sets x_e to true for all $e \in \mathcal{C}$, to false for all events $e \notin \mathcal{C}$ and satisfies ϕ. As the `Explain` prefix is an occurrence net we can apply this technique for analysing this prefix.

Figure 4.15 shows an example for the construction algorithm. We apply the algorithm on the shown occurrence net. The algorithm loops over all conditions and adds clauses. For each condition $c \in \{c_1 \dots c_9\}$, the formula therefore contains a conjunction of those added clauses, $\mathbf{sat}(c)$. Each clause is a disjunction of literals, i.e. negated or not negated variables (the formula $(a \Rightarrow b)$ represents disjunction $(\neg a \vee b)$). The final CNF-formula ϕ is then $\mathbf{sat}(c_1) \wedge \ldots \wedge \mathbf{sat}(c_9)$. Each satisfying assignment represents a configuration. For example we have the satisfying assignment $c_2, e_1, e_4, c_7 = \mathit{true}$ and all other variables set to false. This represents the configuration $\mathcal{C} = \{e_1, e_4\}$ with marking $M(\mathcal{C}) = \{c_2, c_7\}$.

$\mathbf{sat}(c_1)$:
$(e_1 \Rightarrow \neg c_1) \wedge$
$(e_1 \vee c_1)$

$\mathbf{sat}(c_2)$:
$(e_2 \Rightarrow \neg c_2) \wedge$
$(e_2 \vee c_2)$

$\mathbf{sat}(c_3)$:
$(e_3 \Rightarrow \neg e_4) \wedge (e_3 \Rightarrow e_1) \wedge$
$(e_3 \Rightarrow \neg e_5) \wedge (e_4 \Rightarrow e_1) \wedge$
$(e_4 \Rightarrow \neg e_3) \wedge (e_5 \Rightarrow e_1) \wedge$
$(e_4 \Rightarrow \neg e_5) \wedge (e_3 \Rightarrow \neg c_3) \wedge$
$(e_5 \Rightarrow \neg e_3) \wedge (e_4 \Rightarrow \neg c_3) \wedge$
$(e_5 \Rightarrow \neg e_4) \wedge (e_5 \Rightarrow \neg c_3) \wedge$
$((e_3 \vee e_4 \vee e_5) \vee c_3 \vee \neg e_1) \wedge$
$(c_3 \Rightarrow e_1)$



$\mathbf{sat}(c_5)$:
$(e_6 \Rightarrow e_2) \wedge$
$(e_6 \Rightarrow \neg c_2) \wedge$
$((e_6) \vee \neg c_5 \vee \neg e_2) \wedge$
$(c_5 \Rightarrow e_2)$

$\mathbf{sat}(c_4)$:
$(e_5 \Rightarrow e_2) \wedge$
$(e_5 \Rightarrow \neg c_4) \wedge$
$((e_5) \vee \neg c_4 \vee \neg e_2) \wedge$
$(c_4 \Rightarrow e_2)$

$\mathbf{sat}(c_6)$:
$(c_6 \vee \neg e_3) \wedge$
$(c_6 \Rightarrow e_3)$

$\mathbf{sat}(c_7)$:
$(c_7 \vee \neg e_4) \wedge$
$(c_7 \Rightarrow e_4)$

$\mathbf{sat}(c_8)$:
$(c_8 \vee \neg e_5) \wedge$
$(c_8 \Rightarrow e_5)$

$\mathbf{sat}(c_9)$:
$(c_9 \vee \neg e_6) \wedge$
$(c_9 \Rightarrow e_6)$

Figure 4.15: Example for occurrence net to SAT formula translation

The `Explain` prefix usually contains many configurations satisfying $\mathit{proj}(\mathcal{C}) \sqsubset_c \mathit{proj}(O)$, while we are interested in those satisfying $\mathit{proj}(\mathcal{C}) = \mathit{proj}(O)$. So we transform ϕ into a formula ψ whose models are exactly these configurations. For this, we define for each component $\mathcal{A}_i$ the set of events E_i of `Explain` that contain an event e iff $\mathit{proj}(O)_i = \mathit{proj}([e])_i$. Then, we add to ϕ a formula ϕ' expressing that the configuration must contain at least one event of each of the sets $E_1, \ldots, E_n$ and let $\psi = \phi \wedge \phi'$. The models of ψ are the configurations $\mathcal{C}$ of `Explain` such that $\mathit{proj}(\mathcal{C}) = \mathit{proj}(O)$. For more specific queries, we can add further clauses to ψ.

4.3.2 Implementation of the Proactive Diagnosis

We consider discrete time event systems and so we basically have no notion of time. However, an external observer can record emitted observable system events with respect to his global clock. This opens the possibility to do diagnosis "online", i.e. while the system is running. Moreover, we can implement the approach proactively. We already introduced this concept; Figure 4.6 compares reactive and proactive diagnosis. For short, while in the reactive diagnosis we only explain what happened so far, in proactive diagnosis we also speculate about the systems future. With that we might generate behaviour that is not compatible with the final observation. After an alarm has been observed, possibly also while observations are iteratively received, these incompatible behaviours have to be removed.

However, designing and implementing the reactive diagnosis with garbage collection as core component is a difficult task and subject of this section.

For efficiency reasons, we only consider the projection approximation for generating the `Explain` prefix. Note that the definition of O-compatibility (and the approximations O-soundness and projection compatibility) are aimed at extending the prefix only with O-compatible events.

During the diagnosis we do not have complete information on the observation O. We are only given the partial observation $O' \sqsubset O$. An event that is not O'-compatible might be O-compatible, so we cannot use $\mathbf{pe}_{\mathbf{projection}}(U_p)$ for computing possible extensions. So we modify the criteria for accepting events in the unfolding.

Let $\mathit{proj}(A) = \langle \mathit{proj}_1(A), \ldots, \mathit{proj}_n(A)\rangle$ and $\mathit{proj}(B) = \langle \mathit{proj}_1(B), \ldots, \mathit{proj}_n(B)\rangle$. We denote by $\mathit{proj}(A) \bowtie_c \mathit{proj}(B)$ that for each pair of label sequences $(\mathit{proj}_i(A), \mathit{proj}_i(B))$ with $1 \leq i \leq n$, either $\mathit{proj}_i(A) \sqsubseteq \mathit{proj}_i(B)$ or $\mathit{proj}_i(B) \sqsubseteq \mathit{proj}_i(A)$ holds.

Let t be a time variable, with $t = 0$ at the timepoint of the beginning of the diagnosis and $t = t_f$, the timepoint when an alarm is observed, i.e. the observation is complete. The observation is growing over time, we have the invariant $O(t) \sqsubseteq O$ for all timepoints t. Moreover, $O(t) = O$ for all $t \geq t_f$.

We replace the check "$\mathit{proj}(e) \sqsubseteq_c \mathit{proj}(O)$" in $\mathbf{pe}_{\mathbf{projection}}(U_p)$ by "If $t < f_t$ then check $\mathit{proj}(e) \bowtie_c \mathit{proj}(O(t))$ else check $\mathit{proj}(e) \sqsubseteq_c \mathit{proj}(O)$", resulting in $\mathbf{pe}_{\mathbf{proactive}}(U_p)$.

Let $\texttt{Explain}_\mathrm{r}$ and $\texttt{Explain}_\mathrm{p}$ be the branching processes we obtain by applying diagnosis with possible extension function $\mathbf{pe}_{\mathbf{projection}}(U_p)$ and

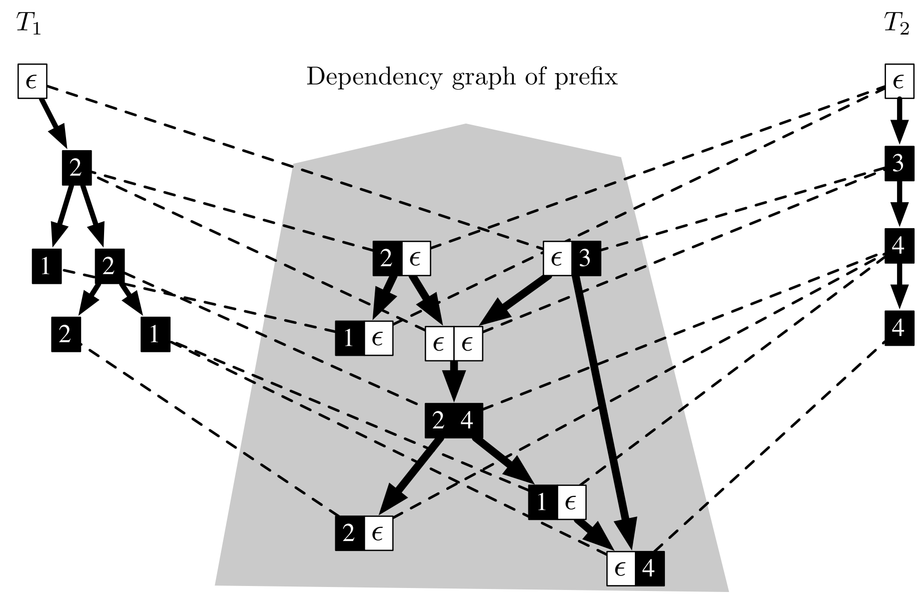


Figure 4.16: Visualization of an example of the used data structure

$\mathbf{pe}_{\mathbf{proactive}}(U_p)$, respectively. For all sequences A and B, if $\mathit{proj}(A) \sqsubseteq_c \mathit{proj}(B)$, then for all $B' \sqsubseteq B$, $\mathit{proj}(A) \bowtie_c \mathit{proj}(B')$ holds. Therefore, if $\mathit{proj}(e) \sqsubseteq_c \mathit{proj}(O)$ then $\mathit{proj}(e) \bowtie_c \mathit{proj}(O(t))$ for every $O(t) \sqsubseteq O$, and so $\mathbf{pe}_{\mathbf{projection}}(U_p)$ is a subset of $\mathbf{pe}_{\mathbf{proactive}}(U_p)$, which implies $\mathtt{Explain}_{\mathrm{r}} \sqsubseteq \mathtt{Explain}_{\mathrm{p}}$. So we can extract $\mathtt{Explain}_{\mathrm{r}}$ from $\mathtt{Explain}_{\mathrm{p}}$ by pruning.

For the efficient implementation of this “garbage collection”, being able to quickly identify events of the unfolding that have to be removed is an important issue.

The decision whether or not an event e is removed from the prefix is based on the projection $\mathit{proj}(e) = \langle \mathit{proj}_1(e), \ldots, \mathit{proj}_n(e)\rangle$. We do not store this tuple for each event explicitly, but keep a tree for each transition system T_i where $i \in \{1, \ldots, n\}$. A sequence $\mathit{proj}_i(e)$ with $i \in \{1, \ldots, n\}$ is represented as a pointer to a node in this tree, i.e. the path to this pointer in the tree is the sequence $\mathit{proj}_i(e)$. Additionally, a pointer in the other direction is kept allowing us to determine all events having some given projection.

For an example, see Figure 4.16. The dependency graph shows only the causality relation between events in an unfolding prefix. Information about conflicts and concurrency is removed, but also not needed for this example. The unfolding is constructed from the labelled synchronized product of transition systems T_1 and T_2. Only labels of events are shown, all events are visible except the only invisible event labelled with $\langle \epsilon, \epsilon \rangle$. The trees for T_1 and T_2 are shown on the right and left of the dependency graph. Consider the two pointers for the event e labelled with $\langle \epsilon, 4 \rangle$. They point to nodes 1 and 4 in the trees of T_1 and T_2 respectively. We have the tree paths $\langle \epsilon, 2, 2, 1 \rangle$ and $\langle \epsilon, 3, 4, 4 \rangle$, where ϵ stands for nothing observed yet, so we have, $proj(e) = \langle \langle 2, 2, 1 \rangle, \langle 3, 4, 4 \rangle \rangle$.

It is easy to see that with such a data structure for storing projections we are able to efficiently prune the prefix based on a given observation. Assume we have observed nothing yet and the current unfolding is the one shown in Figure 4.16. If we observe $O(t) = \langle \langle \epsilon, 3 \rangle, \langle 2, \epsilon \rangle \rangle$, we can remove all events linked by side branches of path $\langle \epsilon, 2 \rangle$ in the tree of T_1, namely the leftmost event with the label $\langle 1, \epsilon \rangle$. As there is no side branch of path $\langle \epsilon, 3 \rangle$ in the tree of T_2, nothing more is removed. If this observation is final, i.e. $O = O(t)$, we can additional remove the linked events of the successors of these paths. In this case, the final prefix consists of the three topmost events with labels $\langle 2, \epsilon \rangle, \langle \epsilon, \epsilon \rangle$ and $\langle \epsilon, 3 \rangle$.

4.3.3 Case Study

Sokoban (see e.g. [10]) is a popular puzzle game, where a single player (the agent) pushes boxes through a maze, trying to push all boxes onto marked target positions. As it is not allowed to pull boxes, they can get stuck in some position, rendering the puzzle unsolvable.

We consider a slightly modified version of this puzzle game, where multiple agents are allowed, that can concurrently move to new fields and push boxes. It can be seen as an abstract model of an Automated Guided Vehicles system as used in warehouses. We define a playing field instance as a rectangular set of two types of cells: normal cells and wall cells. In Figure 4.17, a 3×3 instance with two wall cells (hatched cells) is shown. Agents can move in all four directions of the compass, possibly pushing boxes. Wall cells are not part of the playing field. Consider for example the moves in this figure. In a first step, Agent 1 moves down and Agent 2 moves up (in parallel). Next Agent 3 moves left and in a last step Agent 1 moves right, pushing a box.

We use diagnosis to solve the following problem. Initially, we know the number and identities of the boxes and pushers in the field, but not their positions. When an agent moves, a sensor tells us the identity of the moving

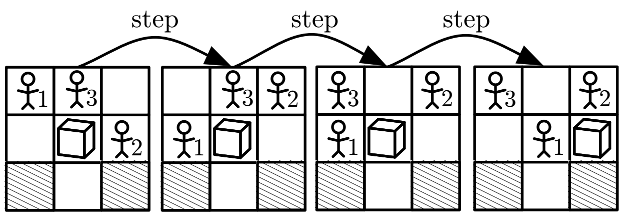


Figure 4.17: Example Sokoban instance with moves

agent and the direction of the move. At each moment in time, a sensor can report an alarm. The task is to determine the possible initial positions of pushers and boxes compatible with the observed sequence of moves. We proceed as follows. First of all we present the model of the game; a product of labelled transition systems. Then we briefly comment on its unfolding and then discuss the used diagnosis procedure. Finally we report on results.

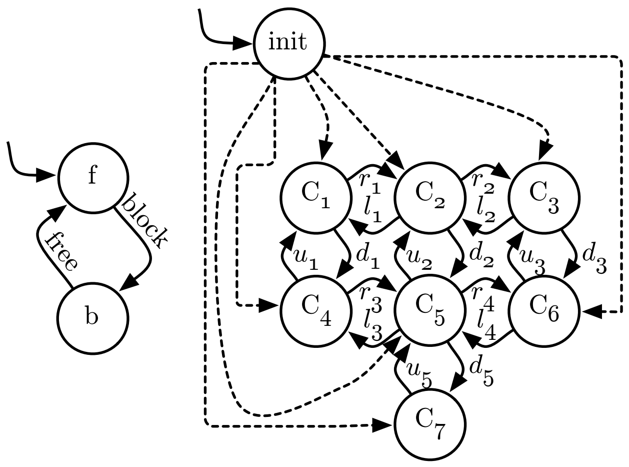


Figure 4.18: Sokoban automaton example

Model

We model the game as a product of two different classes of transition systems: a class (see Figure 4.18 right) modelling the movements of the agents or boxes, and a class (see Figure 4.18 left) modelling a single cell of the playing field. The labelled transition product therefore is of the form

$$\mathbf{A} = \langle \mathcal{A}_{A_1}, \cdots, \mathcal{A}_{A_n}, \mathcal{A}_{B_1}, \cdots, \mathcal{A}_{B_m}, \mathcal{A}_{C_1}, \cdots, \mathcal{A}_{C_k} \mathbf{T}, \ell \rangle$$

and contains a transition system for each agent $A_1, \ldots, A_n$, each box $B_1, \ldots, B_m$ and each cell $C_1, \ldots, C_k$. The global transitions of the synchronization constraint $\mathbf{T}$ model the possible moves of the Sokoban game. We have initialization moves, agent moves, and box moves:

- Initialize - initialize agent A_α or box B_β to C_γ:

$$\mathbf{t} = \left\langle \epsilon, \ldots, \epsilon, \underbrace{\text{init}\, C_\gamma}_{\mathcal{A}_{A_\alpha} \text{or} \mathcal{A}_{B_\beta}}, \epsilon, \ldots, \epsilon, \underbrace{\text{block}}_{\mathcal{A}_{C_\gamma}}, \epsilon, \ldots \epsilon \right\rangle$$

- Agent move - move agent A_α from cell C_β to cell C_γ, where C_γ is to the right of C_β, and there is a transition r_δ from state C_β to state C_γ in A_α: $\mathbf{t} =$

$$\left\langle \epsilon, \ldots, \epsilon, \underbrace{r_\delta}_{\mathcal{A}_{A_\alpha}}, \epsilon, \ldots, \epsilon, \underbrace{\text{free}}_{\mathcal{A}_{C_\beta}}, \epsilon, \ldots, \epsilon, \underbrace{\text{block}}_{\mathcal{A}_{C_\gamma}}, \epsilon, \ldots \epsilon \right\rangle$$

  Analogous for moves in other directions.

- Box move - agent A_α at position C_β pushes box B_ρ at position C_γ to position C_p. C_β is the left neighbour of C_γ, which is the left neighbour C_p. A_α has a transition r_δ from state C_β to state C_γ, and B_ρ has a transition r_σ from state C_γ to state C_p:

$$\mathbf{t} = \left\langle \epsilon, \ldots, \epsilon, \underbrace{r_\delta}_{\mathcal{A}_{A_\alpha}}, \epsilon, \ldots, \epsilon, \underbrace{r_\sigma}_{\mathcal{A}_{B_\rho}}, \epsilon, \ldots, \epsilon, \underbrace{\text{free}}_{\mathcal{A}_{C_\beta}}, \epsilon, \ldots, \right.$$
$$\left. \epsilon, \underbrace{\text{block}}_{\mathcal{A}_{C_\gamma}}, \epsilon, \ldots \epsilon \right\rangle$$

  Analogous for other directions.

We now define the labelling function. For every local transition $t \in T_i$ with $i \in \{A_1, \ldots, A_n\} \cup \{B_1, \ldots, B_m\}$: if t is one of r_j, l_j, u_j, o_j for some number j, then $\ell(t)$ is the respective symbol from r, l, u, o. Moreover, for every $t \in T_i$ with $i \in \{C_1, \ldots, C_n\}$ we set $l_i(t) = \epsilon$. Intuitively, the information we obtain from the occurrence of a global transition is only the agent (and box) participating in the transition, and the direction of movement. We do not

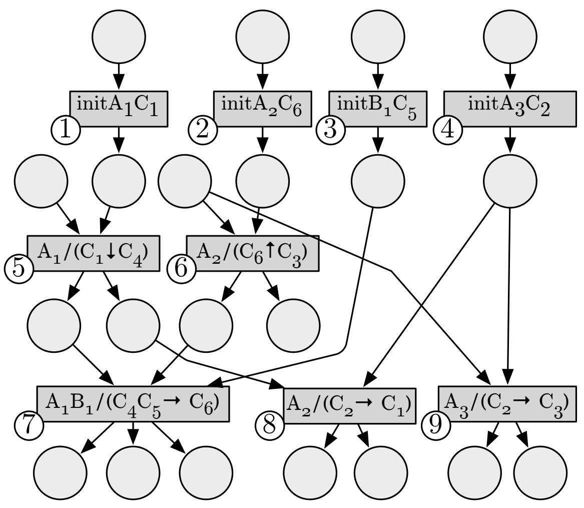


Figure 4.19: Part of the Sokoban unfolding

obtain information about which C_i's participate in the global transition (this would correspond to the sensor being able to report the current position of the agent).

Unfolding

We now consider the unfolding of the Petri net representation of the labelled product. We use $[A_1/(C_2 \rightarrow C_3)]$ to denote a global transition that moves agent A_1 from C_2 to C_3 (to the right), and $[A_1B_2/(C_3C_4 \rightarrow C_5)]$ for the transition where A_1 at position C_3 pushes box B_2 at position C_4 to position C_5 (to the right), and $[A_1\text{init}C_2]$ for the transition that initializes A_1 to position C_2. We use a similar notation for labels. For example $\ell([A_1/(C_2 \rightarrow C_3)]) = [A_1 \rightarrow]$, or $\ell([A_1B_2/(C_3C_4 \rightarrow C_5])) = [A_1B_2 \rightarrow]$. Figure 4.19 shows a small part of the unfolding of our running example. The set of events {①, ②, ③, ④} can fire in any order, resulting in the initial positions shown in the leftmost board in Figure 4.17. If we fire {①, ②, ③, ④, ⑤, ⑥, ⑦, ⑧} we get the rightmost board in Figure 4.17. The causal order shows, that we cannot fire ⑧ unless we fired ④ and ⑤. We cannot add ⑨ to this configuration, because it is in conflict with ⑥.

Experiments

Our implementation of reactive and proactive diagnosis uses the Mole unfolder [45] and the Minisat SAT-solver [12]. We use Mole to generate the prefix. Garbage collection manipulates Mole's data structures directly. We also change the order of events in the possible extension queue, allowing us to first unfold events that fulfil the check for reactive diagnosis, otherwise almost all events are removed during garbage collection.

We apply our implementation to the Sokoban example. Table 4.1 shows experiments[2] on five random Sokoban instances. For *reactive diagnosis* only the first three columns are relevant. For each Sokoban instance a few attributes are given: (A) number of agents, (B) number of boxes, (W) number of walls, (Dim) dimensions of the playing field, number of places (P) and transitions (T) of the Petri net model. For each instance, we execute the reactive diagnosis for three observations with lengths 3, 10 and 15, generated randomly (random start configuration and random simulation). Reactive diagnosis yields each time an `Explain` prefix, whose number of events is shown below the sequence length. Our algorithm can generate prefixes with tens of thousands of events in seconds/a few minutes. The unfolding is started after the error alarm occurs. Column 3 gives the time in seconds since the error alarm occurs until the construction of `Explain` terminates.

As already discussed, we trade the construction of a prefix that only contains compatible events for a more efficient construction that can however produce *spurious explanations*. Consider for example Figure 4.20, playing field (I), with $O = \langle (A_1 \leftarrow), (A_1 \leftarrow), (A_1 \downarrow), (A_2 \rightarrow), (A_1 \downarrow) \rangle$. The only initial

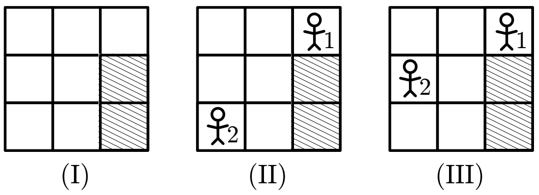


Figure 4.20: Spurious explanation example

positions from which this sequence can be observed is shown in (II). In (III)

[2]All experiments were realized on a Sun Fire X4600 workstation, featuring 8× Dual-Core AMD Opteron CPUs and 64 GB memory.

Table 4.1: Running times of the re- and proactive algorithm

	#Seq/#E	React	Pro/2s	Pro/5s	Pro/10s	Pro/20s
Instance 1	3	0.2s	0.6s	1.7s	4.5s	8.4s
A:3 B:4	373e		+373e	+373e	+373e	+373e
W:4	10	1.5s	1.4s	3.4s	6.2s	11.5s
Dim:6×7	2427e		+2427e	+2427e	+2427e	+2427e
Pl:343	15	5.7s	2.5s	2.8s	8.5s	6.2s
Tr:1592	8397e		+7615e	+8103e	+8397e	+7991e
Instance 2	3	0.7s	0.7s	2.0s	4.6s	9.6s
A:6 B:6	525e		+525e	+525e	+525e	+525e
W:9	10	6.3s	5.1s	4.5s	4.3s	6.1s
Dim:8×5	4543e		+1423e	+1796e	+2569e	+2387e
Pl:532	15	65.9s	61.7s	64.7s	65.7s	54.1s
Tr:2520	30251e		+3181e	+4435e	+6800e	+13150e
Instance 3	3	0.3s	1.3s	4.1s	10.0s	19.1s
A:7 B:3	381e		+381e	+381e	+381e	+381e
W:10	10	5.4s	3.4s	2.7s	4.4s	9.8s
Dim:8×5	4124e		+1344e	+2349e	+4124e	+4124e
Pl:450	15	421.6s	413.1s	367.4s	363.7s	387.8s
Tr:1798	92156e		+2899e	+6493e	+11741e	+20522e
Instance 4	3	0.8s	0.9s	1.6s	3.5s	9.2s
A:6 B:5	777e		+777e	+777e	+777e	+777e
W:10	10	21.2s	17.5s	16.3s	15.1s	15.4s
Dim:7×9	7677e		+1934e	+2674e	+2958e	+3769e
Pl:767	15	372.6s	379.4s	374.4s	376.9s	361.1s
Tr:5011	68671e		+2423e	+3285e	+4120e	+6219e
Instance 5	3	1.5s	0.8s	1.9s	4.3s	10.9s
A:6 B:3	1104e		+1104e	+1104e	+1104e	+1104e
W:2	10	121.1s	112.7s	121.0s	111.5s	114.3s
Dim:9×8	32406e		+2150e	+2506e	+2621e	+2663e
Pl:729	15	2991.7s	3052.9s	2896.3s	2934.7s	2960.6s
Tr:5622	211492e		+2207e	+2953e	+3291e	+3268e

the third move of the observation sequence cannot be executed, as A_2 is in the way for A_1 to move down. However, our algorithm also returns (III) because from there, another observation O' is possible such that $f(O) = f(O')$. This observation is, e.g. $O' = \langle (A_1 \leftarrow), (A_1 \leftarrow), (A_2 \rightarrow), (A_1 \downarrow), (A_1 \downarrow) \rangle$. It has the same projection to transition systems than O.

The next experiment analyzes the precision of the algorithm on our example. For each Sokoban instance, we generate `Explain` and use the SAT solving based method presented in Section 4.3.1 to compute the initial positions of all the configurations $\mathcal{C}$ of `Explain` such that $f(\mathcal{C}) =_c f(O)$. Then, for each initial position we check whether the observation is indeed executable. In Table 4.2, for each `Explain` prefix and observations of lengths 5, 10, 20, 30, 40 and 60 respectively, we record the amount of spurious (S) and correct (C) configurations. Most of the time, the number of correct explanations exceeds the number of spurious ones. Since checking spuriousness of an explanation is relatively fast, we conclude that the gain of efficiency compensates for the loss in precision.
Results for proactive diagnosis are shown in Table 4.1, Columns 4-7. Each column corresponds to a different speed of the system, in which a move occurs (or the error alarm triggers once) every 2, 5, 10 or 20 seconds, respectively. The unfolder is started at the same time as the system, and "constructs ahead", anticipating the next possible moves. Whenever a new event is added, the garbage collector is invoked to remove events corresponding to false guesses. For each instance and system speed we provide the time elapsed since the error alarm is triggered until `Explain` is constructed, and the number of events the proactive diagnosis is "ahead" of the reactive diagnosis (i.e. the number of events that the unfolder has already constructed and are not removed later by garbage collection) when the error alarm is triggered. The results do not support the adoption of proactive diagnosis as the standard for the Sokoban example, although it outperforms reactive diagnosis in a number of cases. The worst case for proactive diagnosis happens when the garbage collection from the previous round is not yet finished, when a new observation is made. This is for example the reason why in the first experiment, with sequence length 15, proactive diagnosis with system speed 10 is more events ahead than proactive diagnosis with speed 20.

Table 4.2: Correct and spurious start configurations

Instance	T	#5	#10	#20	#30	#40	#60
A:2 B:2 W:8 Pl:179	C	37520	459	357	6	4	3
Dim:5×7 Tr:444	S	3780	76	70	3	2	1
A:2 B:3 W:2 Pl:101	C	44640	23472	42	6	6	6
Dim:4×4 Tr:278	S	4950	2880	40	0	0	0
A:2 B:3 W:5 Pl:149	C	26664	2430	672	420	60	60
Dim:6×4 Tr:371	S	3432	3300	504	336	210	0
A:3 B:3 W:7 Pl:195	C	12436	1478	92	6	6	6
Dim:3×9 Tr:438	S	6396	1806	42	42	42	0
A:2 B:1 W:5 Pl:123	C	2403	475	60	44	36	9
Dim:6×5 Tr:283	S	80	0	15	14	36	20
A:3 B:1 W:4 Pl:154	C	89880	45714	174	16	3	1
Dim:6×5 Tr:482	S	13398	6780	142	27	10	0

4.4 Conclusion

We have presented an unfolding based algorithm for the diagnosis problem, following the ideas of [3] and have proven a soundness and completeness result: the finite prefix constructed by the presented algorithm is O-sound and O-complete for a given observation O. Since the construction of this prefix is algorithmically difficult, we have presented an algorithm that constructs an overapproximation. For this, we make crucial use of our model of computation: products of transition systems. While equivalent to Petri nets in expressive power, the additional structure of these products can be put to good use to improve the efficiency of the routine for the generation of possible extensions.

We have implemented our algorithm, and reported on some experiments. The unfolder is very fast (the performance of the very efficient, general purpose unfolder Mole is not impaired), and yet of reasonable precision. The presented diagnosis algorithm is connected to a SAT solver, allowing to efficiently extract information about the set of explanations. Finally, we have implemented a proactive diagnosis algorithm that runs online together with the system and speculates about the future of the system.

Chapter 5

Computing the Reveals Relation

In the research area of diagnosis we consider partial observable systems that can fail. A failure is an irreversible undesired state of the system. Failures are preceded by faults that may be reversible. Alarms indicate possible faults. Partial observability is usually modelled using sensors attached to parts of the system. An observer records a sequence of visible system events, the observation. Different diagnosis problems have been studied. We identify two major problems: (1) given the observation of a run in a partially observable system that contains invisible faults, decide if a fault has occurred and (2) given a partially observable system that triggers visible alarms, check if a fault has occurred and, if the answer is positive, give an explanation for it.

In both cases we have to reason about the occurrence of invisible events using only the given information about the occurrence of visible events. That is where the reveals relation comes into play. We say, loosely speaking, that event a *reveals* b iff, whenever a occurs, event b has already occurred or will eventually occur. Assuming a is visible and b not, from the visible occurrence of a we can infer the invisible occurrence b.

In Figure 5.1 we illustrate the reveals relation with an example. It shows the playing field of a Sokoban game where agents and boxes are allowed to move in parallel. We start with the initial situation.

Denote with $\langle A(2\alpha), Z(2\beta)\rangle \to \langle A(2\beta), Z(2\gamma)\rangle$ the move (Move3 in the figure) where Agent A at position 2β pushes box Z at position 3β to positions 2β and 2γ, respectively. If this move is applied, in a next step we can apply either move $\langle A(4\gamma), Y(5\gamma)\rangle \to \langle A(5\gamma), Y(6\gamma)\rangle$ (Move4, not shown) or move $\langle B(6\alpha), X(6\beta)\rangle \to \langle B(6\beta), X(6\gamma)\rangle$ (Move2). Because both moves can be

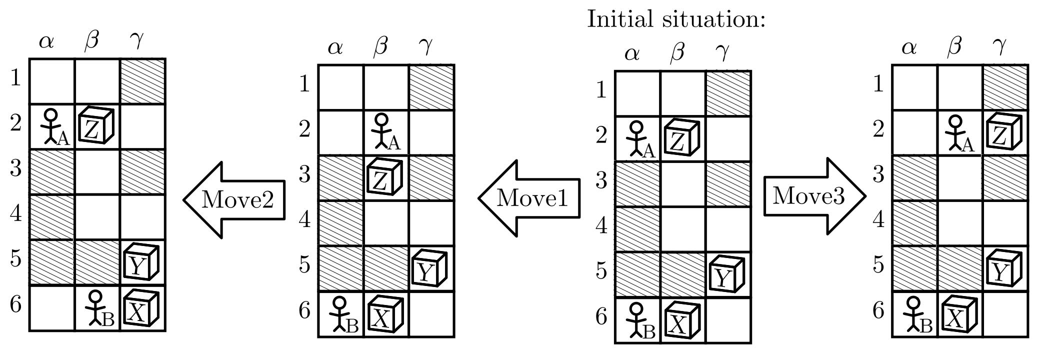


Figure 5.1: Example for appliance of reveals - parallel Sokoban moves

applied, Move3 does not reveal any of them. Note that we only describe box moves and not the single agent moves needed to reach the right push positions.

Consider a second experiment. We start again with the initial situation and apply move $\langle A(1\beta), Z(2\beta)\rangle \rightarrow \langle A(2\beta), Z(3\beta)\rangle$ (Move1). This finally renders position 4γ unreachable for any agent, however, reaching this field is necessary to push Y to field 6γ. Move2 is the only move that can push something to this field and no action can deactivate this move. Assuming that an enabled move cannot be withheld forever, B will eventually push X to this field (Move2). So, Move1 reveals Move2. Assume we only put a sensor on field 3β and everything else is invisible. When the sensor registers movement, we immediately know that Move2 will eventually occur or has already occurred.

The reveals relation itself was introduced in 2009 by Stefan Haar [24] and defined on a special kind of Petri nets; occurrence nets. Moreover, decidability of the question "given to events a and b, does a reveal b" was shown. However, the results cannot be used to practically compute this relation as the derived approach has very high complexity.

In this chapter we notably improve the decidability results for reveals. Moreover, we utilize these results to formulate and implement a practical algorithm for actually computing this relation.

We proceed as follows. In Section 5.1, we give basic definitions and results on the reveals relation. In Section 5.2, we improve the decidability results for reveals. Then, in Section 5.4.1, we present the algorithm for computing reveals. In Section 5.4.2 we show an efficient implementation for computing reveals, give experimental data on the presented decidability results and report on experiments. Then, we conclude in Section 5.5.

5.1 The "Reveals" Relation

Recall the definition for occurrence nets in Section 3.2.2 and the definition of a fair run in Section 3.2.1. Let x and y be events of some occurrence net. Informally, x reveals y iff, whenever x happens in a fair run, y will eventually happen or has already happened.

In Section 3.2.1 we defined a *fair run*. Informally, a fair run cannot delay firing an enabled event forever. With that, we define reveals:

Definition 27 (Reveals)**.** *We say that x* reveals *y ($x \rhd y$) iff any fair run that contains event x also contains event y.*

Trivially, if $y < x$ holds for two events x and y then any fair run σ with $x \in \sigma$ also contains y as y has to fire before x fires. So we have $x \rhd y$.

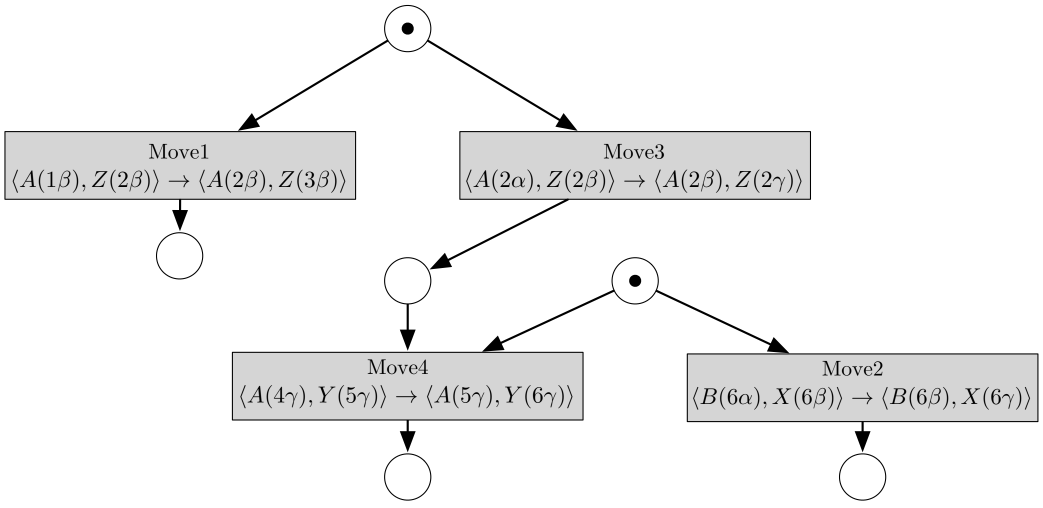

Move1
$\langle A(1\beta), Z(2\beta)\rangle \to \langle A(2\beta), Z(3\beta)\rangle$
Move3
$\langle A(2\alpha), Z(2\beta)\rangle \to \langle A(2\beta), Z(2\gamma)\rangle$
Move4
$\langle A(4\gamma), Y(5\gamma)\rangle \to \langle A(5\gamma), Y(6\gamma)\rangle$
Move2
$\langle B(6\alpha), X(6\beta)\rangle \to \langle B(6\beta), X(6\gamma)\rangle$

Figure 5.2: Occurrence net example for reveals

Figure 5.2 shows a simplified Petri net model (that is also an occurrence net) of the Sokoban game from Figure 5.1. We again verify that Move1 $\rhd$ Move2 holds: a run containing Move1 cannot contain Move3 because Move1 is in conflict with Move3 and no run can contain conflicting events. This run also cannot contain Move4 because Move3 is a causal predecessor of Move4.

So any run containing Move1 does not contain Move4. As the run is fair it cannot delay firing an enabled event forever. Move2 is enabled by its precondition and Move4 cannot consume this token and disable Move2 so Move2 has to fire eventually and is therefore contained in the run.

We now consider the slightly more complex occurrence net shown in Figure 5.3.

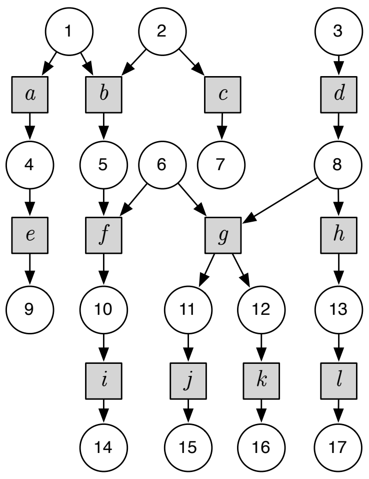


Figure 5.3: Example Occurrence net used for explaining reveals

Note that reveals is not a symmetric relation: because of causality we have $h \rhd d$. On the other hand, $d \rhd h$ does not hold because there exists a fair run σ with $d \in \sigma$ and $g \in \sigma$, but then we have $h \notin \sigma$.

We now consider the more involved problem "$i \rhd l$?". Assume some fair run σ with $i \in \sigma$. Then, because of causality $f \in \sigma$ holds. As f is in conflict with g we have $g \notin \sigma$ as no run can contain conflicting events. Initial condition 3 enables only event d, so $d \in \sigma$. So either g or h is contained in σ. As event $g \notin \sigma$ we have $h \in \sigma$. As l is the only event enabled by 13 and h marks 13 we have $l \in \sigma$. So for any run σ with $i \in \sigma$ it holds that $l \in \sigma$ so $i \rhd l$. On the other hand $l \rhd i$ does not hold as the run that contains events d, h, l, a cannot contain i as $i\#a$ holds.

In terms of diagnosis if i is visible but l is not, we can conclude from the occurrence of i that l also occurs.

Lemma 8. *(Reveals defined with conflict sets [24]) Let O be an occurrence* *net with events $x, y \in O$. We have following equality:*

$$x \rhd y \Leftrightarrow \#[x] \supseteq \#[y]$$

Proof. By definition we have the equality $x \rhd y \Leftrightarrow (x \in \sigma \Rightarrow y \in \sigma$ for all fair runs $\sigma)$. If $x \in \sigma \Rightarrow y \in \sigma$ holds for all fair runs σ then $\#x \supseteq \#y$ holds. Assume the contrary: $x \in \sigma \Rightarrow y \in \sigma$ holds for all fair runs σ and $\neg(\#x \supseteq \#y)$. Then there exists an event z with $z \notin \#x$ and $z \in \#y$, i.e. $\neg(z\#x)$ and $z\#y$. Then $\lceil z \rceil \cup \lceil x \rceil$ is a configuration so there exists a run σ' containing z and x. By assumption, we have $y \in \sigma'$. But, as $z\#y$, this is a contradiction. If $\#x \supseteq \#y$ holds then $x \in \sigma \Rightarrow y \in \sigma$ holds for all fair runs σ. Assume the contrary. Then there exists run σ'' with $x \in \sigma''$ and $\neg(y \in \sigma'')$. So there exists event z with with $\neg(z\#x)$ and $z\#y$, otherwise, because of the maximality of σ'', $y \in \sigma''$ holds. But this contradicts $\#x \supseteq \#y$. □

In our running example we have $\#[i] = \{g, k, j, e, c, a\}$ and $\#[l] = \{g, k, j\}$, so $\#[i] \supseteq \#[l]$ holds but $\#[l] \supseteq \#[i]$ does not. Moreover, we now can again confirm $Move1 \rhd Move2$ in Figure 5.2 as #[Move1] = {Move3, Move4} and #[Move2] = {Move4} and with that #[Move1] ⊇ #[Move2].

A set of events in which every event reveals each other event has the property that if one event occurs all other events inevitably also occur. Such a set is called a facet:

Definition 28 (Facet)**.** *A set of events F is a* facet *iff $a \rhd b$ holds for all events $a, b \in F$.*

As all events of a facet occur if only one does we can summarize and abstract it by “merging” its events into one single event. In Figure 5.3 we can identify four facets, namely $F_1 = \{a, c, e\}$, $F_2 = \{g, j, k\}$, $F_3 = \{h, l\}$ and $F_4 = \{f, i\}$. The abstracted occurrence net is shown in Figure 5.4.

Until now, we only considered the reveals relation in finite occurrence nets. If we consider infinite occurrence nets, deciding reveals involves considering infinite conflict sets. Decidability results and algorithms therefore are subject of the next sections.

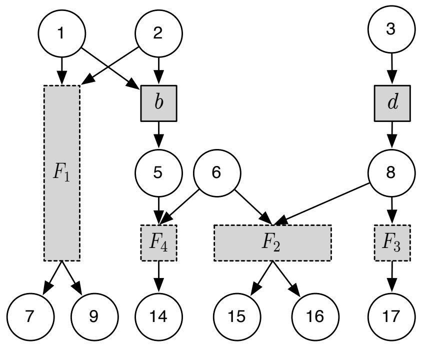


Figure 5.4: Facet abstraction of Occurrence net

5.2 Existing Bound for Deciding the *Reveals* Relation

Let $N = (P, T, F, M_0)$ be a (finite) net and let $U = (C, E, G, C_0)$ be its unfolding. We deal with the following problem: given two events $x, y \in E$ decide whether $x \rhd y$ holds. Remember that $x \rhd y$ is equivalent to $\#[x] \supseteq \#[y]$. Although the sets $\#[x]$ and $\#[y]$ can be infinite this problem is decidable, as shown in [24]. We briefly discuss that result.

Let A and B be two arbitrary sets of elements. If $A \not\supseteq B$ then there exists an element z such set z is contained in A but not in B, otherwise B would be a subset of or equal to A. So z is a "witness" for the property $A \not\supseteq B$. Let us carry this to the problem of deciding reveals.

Definition 29 (Witness)**.** *If x does not reveal y, i.e. $\neg(\#x \supseteq \#y)$, then there exists a not-reveals* witness *z, such that $z\#y$ and $\neg(z\#x)$. Shortly, we write* $\mathbf{wit}(\neg(x \rhd y), z)$*.*

So when an event a does not reveal another event b then there exists an event z that is a witness for this. For the presentation of the result we need the definition of the height of an event. Informally the *height* of an event e is the longest possible chain of events, with respect to $<$, in the local configuration of $\lceil e \rceil$.

Definition 30 (Height of event)**.** *The height $\mathcal{H}(e)$ of event e is defined recursively. Let P_e be the set of pre-events ${}^\bullet({}^\bullet e)$. If $P_e = \varnothing$ then $\mathcal{H}(e) = 1$. Otherwise, let $e' \in P_e$ be an event so that $\forall\, e'' \in P_e : \mathcal{H}(e') \geq \mathcal{H}(e'')$. Then $\mathcal{H}(e) = 1 + \mathcal{H}(e')$.*

We naturally extend the definition of the height of an event to configurations and prefixes: let E be the set of events of some configuration $\mathcal{C}$ or some prefix $O[E]$. The height of $\mathcal{C}$ or $O[E]$ is the height of an event with maximum height in the set E. The result for the bound of reveals is now as follows:

Lemma 9 (Bound for reveals [24]). *If $\neg(x \rhd y)$ then there exists a reveals witness z with bounded height.*

For deciding reveals, one can then use following algorithm: let h be the height bound for witness z, i.e. if $\neg(x \rhd y)$ then there exists z so that $\mathbf{wit}(\neg(x \rhd y), z)$ and $\mathcal{H}(z) < h$. If the prefix is finite we check $\neg(\#x \supseteq \#y)$ and are done. If the prefix is not finite we construct the contained finite prefix that only contains all events up to height h and then we proceed as in the case for the finite prefix.

Unfortunately, the height bound of z is very huge, so it is unlikely that an efficient algorithm can be derived. In the following section we give an improved height bound that gives hope to make the reveals relation practically computable.

5.3 Improved Bound for Deciding the *Reveals* Relation

As in the previous section, let $N = (P, T, F, M_0)$ be a (finite, 1-safe) net and let $U = (C, E, G, C_0)$ be its unfolding. As already discussed in the last section, if x does not reveal y there exists a not-reveals witness z that has bounded height. In this section we give a better bound. Roughly estimated and informal, the height of the bound in [24] is n-times the height of the complete unfolding prefix where n is the number of reachable markings of N and the new bound is obtained by replacing n with the constant factor 2, which is a huge improvement.

Recall the definition of a complete prefix in Section 3.2.3. Let $U_1 = (C_1, E_1, G_1, C_{01})$ (level-1 unfolding) be a complete prefix of U, i.e. a finite prefix that contains for each reachable marking M of N a configuration $\mathcal{C}$ so that $Mark(\mathcal{C}) = M$. Let $e \in L_1$ iff there exists event e' with $e' \notin E_1, e \in E$ so that $e \in {}^{\bullet\bullet}e'$. Loosely speaking, L_1 contains the set of events that have successors in U but not in U_1. We now define the Level-2 unfolding of net N.

Definition 31 (Level-2 unfolding)**.** *Let $e \in L_2$ if there exists events e', e'' so that $e'' \in L_1$ and $e'' < e' < e$ with $M_e = M_{e'}$ and e is minimal, i.e. for all events $e''' < e$, there exists no events e', e'' so that $e'' \in L_1$, $e'' < e' < e'''$ and $M_{e'} = M_{e'''}$. Let $L_2^{\leq}$ be the downward closure of L_2 (with respect to $<$). We have $U_2 = (U_1 \bigcup O[L_2^{\leq}])$.*

Note that we define U_2 as the union of the downward closure of $O[L_2^{\leq}]$ and the prefix U_1 and not as just the downward closure of $O[L_2^{\leq}]$, because not all elements of the level-1 unfolding must be contained in the downward closure of $O[L_2^{\leq}]$. This is because in U_1 there might exist maximal elements with respect to $<$, not contained in L_1.

Denote with U^M the unfolding of N with initial marking M. Let $U_<^M$ be the maximal prefix of U^M that does not contain events e so that there exist events e' and e'' with $e' < e'' < e$ and $M_{e'} = M_{e''}$.

Recall Section 3.2.2, Definition 5 for the definition of the postfix $U/_C$ of a configuration C. As U_1 is complete it contains a configuration $\mathcal{C}_M$ for every reachable marking M such that $Mark(\mathcal{C}_M) = M$. Configuration $\mathcal{C}_M$ has the same marking as the minimal conditions of U^M so we have the isomorphism $U/_{\mathcal{C}_M} \cong U^M$.

Lemma 10. *For every reachable marking M, U_2 contains $(U/_{\mathcal{C}_M})_<$, an isomorphic copy of $U_<^M$.*

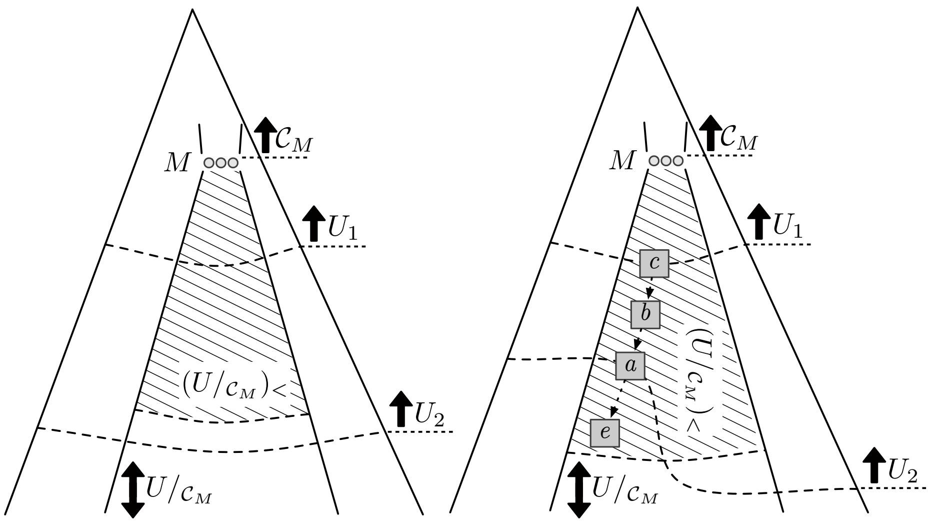


Figure 5.5: Illustration of Lemma 10

Proof. Assume the contrary. Then there exists an event e such that $e \in (U/_{\mathcal{C}_M})_<$ and $e \notin U_2$. From $e \notin U_2$ and the definition of the level-2 unfolding it follows that there exist events a, b, c with $a \in L_2$, $c \in L_1$, $c < b < a < e$ and $M_a = M_b$. Since a prefix is causally closed and $e \in (U/_{\mathcal{C}_M})_<$, for each event $j \leq e$ it holds that either $j \in (U/_{\mathcal{C}_M})_<$ or $j \in \mathcal{C}_M$. As C_M is contained in U_1 and $c \in L_1$ we have $b, a, e \in (U/_{\mathcal{C}_M})_<$. By definition, U^M does not contain an event o so that there exists events o' and o'' with $o' < o'' < o$ and $M_{o'} = M_{o''}$. Then, as $(U/_{\mathcal{C}_M})_<$ is isomorphic to U^M and we have $e \notin (U/_{\mathcal{C}_M})_<$, a contradiction. □

Figure 5.5/left sketches the following Lemma and Figure 5.5/right the assumed contrary. For further use we define $\kappa := \mathcal{H}(U_2)$ to be the height of the level-2 unfolding of U.

Lemma 11. *Let $\mathcal{C}$ be a configuration and let e be an event with $e \in U/_{\mathcal{C}}$. If $\mathcal{H}(e) < \kappa + \mathcal{H}(\mathcal{C})$ then there exist two configurations $\mathcal{C}_1, \mathcal{C}_2$, with $\mathcal{C} \subseteq \mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq (\mathcal{C} \cup \lfloor e \rfloor)$ so that $Mark(\mathcal{C}_1) = Mark(\mathcal{C}_2)$ and $\mathcal{H}(C_1) < \mathcal{H}(C_2)$.*

Proof. $U/_{\mathcal{C}}$ is isomorphic to U^M, with $M = Mark(\mathcal{C})$. Lemma 10 denotes that there exists an isomorphic copy $(U/_{\mathcal{C}_M})_<$ of $U^M_<$ such that $(U/_{\mathcal{C}_M})_<$ is contained in U_2. The height of e implies a chain of events $j = e_1 < \ldots < e_n = e$ for some event $j \in C$ with $n > \kappa$. Let e' be the isomorphic copy of e in $U/_{\mathcal{C}_M}$. We also have such a chain of events in $U/_{\mathcal{C}_M}$. Then, $\mathcal{H}(e') > \kappa$ so $e' \notin U_2$ and also $e' \notin (U/_{\mathcal{C}_M})_<$.

Because $e' \notin (U/_{\mathcal{C}_M})_<$, there exist events a and b with $a, b \in (U/_{\mathcal{C}_M})_<$ so that $M_a = M_b$ and $a < b < e'$, otherwise, due to isomorphism, we have event e'' with $e'' \notin U^M_<$ and $e'' \in U^M$ so that there exist no two events $a', b' \in U^M_<$ with $M_{a'} = M_{b'}$ and $a' < b' < e''$. But then $U^M_<$ is not maximal, as it does not contain e'', a contradiction.

Let a''' and b''' be the events isomorphic to a and b in $\mathcal{C}$. We have $M_{a'''} = M_{b'''}$. Let $\mathcal{C}_1 = \mathcal{C} \cup \lceil a''' \rceil$ and $\mathcal{C}_2 = \mathcal{C} \cup \lceil b''' \rceil$, then we have $\mathcal{C} \subseteq \mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq (\mathcal{C} \cup \lfloor e \rfloor)$ so that $Mark(\mathcal{C}_1) = Mark(\mathcal{C}_2)$ and $\mathcal{H}(C_1) < \mathcal{H}(C_2)$. □

Recall Section 3.2.3. Given sets S_1, S_2 and S_3 with $S_1 \subseteq S_2$ and mapping $f : S_2 \to S_3$. With $f \colon S_1 \leftrightarrow S_3$ we denote that that $f|_{S_1}$ (restriction of the domain of f to S_1) is a bijection. Let $f : (C \cup E) \to (P \cup T)$ be the

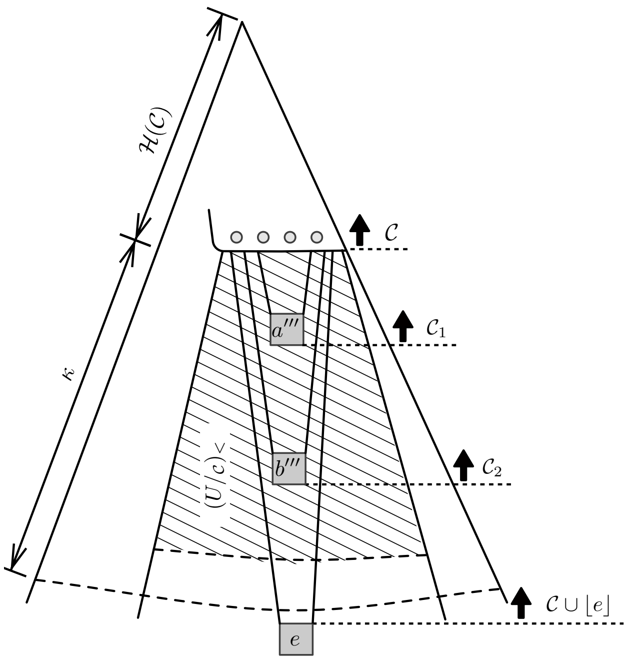


Figure 5.6: Illustration of Lemma 11

function mapping f. Moreover, recall that the $Cut(\mathcal{C})$ of configuration $\mathcal{C}$ are the $<$-maximal elements, defined as $(C_0 \ \cup \ \mathcal{C}^\bullet) \setminus {}^\bullet\mathcal{C}$.

Lemma 12. *For every configuration $\mathcal{C}$ it holds that $f\colon Cut(\mathcal{C}) \leftrightarrow f(Cut(\mathcal{C}))$, i.e. with $(C \cup E)$ restrticted to the conditions of $Cut(\mathcal{C})$, f is a bijection.*

Proof. The proof is by induction. With $\mathcal{C} = \varnothing$ we have $f\colon C_0 \leftrightarrow M_0$ which is equivalent to $f\colon Cut(\mathcal{C}) \leftrightarrow f(Cut(\mathcal{C}))$. Assume $f\colon Cut(\mathcal{C}) \leftrightarrow f(Cut(\mathcal{C}))$ holds for some configuration $\mathcal{C}$ and let e be an event such that $\mathcal{C} \overset{e}{\rightsquigarrow}$, i.e. $\mathcal{C}_{+1} := \mathcal{C} \cup e$ is a configuration. We have $Cut(\mathcal{C}_{+1}) = (Cut(\mathcal{C}) \setminus {}^\bullet e) \cup e^\bullet$. In the net we have that $f(\mathcal{C})$ marks $f(Cut(\mathcal{C}))$ with $f({}^\bullet e) \subseteq f(Cut(\mathcal{C}))$ and then $f(e)$ is enabled. The net is safe so the result of firing $f(e)$ is $(f(Cut(\mathcal{C})) \setminus f({}^\bullet e)) \cup f(e^\bullet)$. As, by definition of a branching process, $f\colon {}^\bullet e \leftrightarrow f({}^\bullet e)$ and $f\colon e^\bullet \leftrightarrow f(e^\bullet)$ and by induction hypothesis $f\colon Cut(\mathcal{C}) \leftrightarrow f(Cut(\mathcal{C}))$ we have $f\colon Cut(\mathcal{C}_+) \leftrightarrow f(Cut(\mathcal{C}_+))$ □

Lemma 13. *Let a, b be conditions contained in the same configuration with $f(a) = f(b)$. One of the following statements holds: $a < b$, $a = b$ or $a > b$.*

Proof. If a **co** b holds for two events a, b with $f(a) = f(b)$ there has to exist a configuration $\mathcal{C}$ with $a, b \in Cut(C)$. Lemma 12 implies that no such configuration exists as otherwise $f\colon Cut(\mathcal{C}) \leftrightarrow f(Cut(\mathcal{C}))$ would not be a bijection. Moreover, $a \# b$ cannot hold, because that would imply that there exist two events $a' = {}^\bullet a$ and $b' = {}^\bullet b$ with $a \# b$ but a and b are contained in the configuration and with that, by definition, not in conflict. As $\neg(a \textbf{ co } b)$ and $\neg(a \# b)$, a and b are causally related or equal.

□

Theorem 2 (Bound for reveals)**.** *Let x, y be events so that $\neg(x \rhd y)$ and let $n = max(\mathcal{H}(x), \mathcal{H}(y))$. Then there exists an event z such that $\mathcal{H}(z) \leq n + \kappa$ and* $\textbf{wit}(\neg(x \rhd y), z)$.

Proof. Let x, y be events and assume $\neg(x \rhd y)$, as stated in the Theorem. Recall that if $\neg(x \rhd y)$ then there exists a witness z with $z \# y$ and $\neg(z \# x)$. Shortly we write $\textbf{wit}(\neg(x \rhd y), z)$.

Assume $(x \# y)$. Then $z := x$ is a witness with height $\mathcal{H}(x)$. As $\mathcal{H}(x) \leq n + \kappa$ we are done. So for now on assume $\neg(x \# y)$. Further let z be minimal, i.e. z is an event so that $\textbf{wit}(\neg(x \rhd y), z)$ holds and for all events $e < z$, $\textbf{wit}(\neg(x \rhd y), e)$ does not hold. So for all such events e, $\neg(x \# e)$ and $\neg(y \# e)$ holds because if $\neg(z \# x)$ no event with $t < z$ can be in conflict with x by definition of a conflict. Moreover, due to this minimal condition there exists a condition b that sets z in direct conflict with y, i.e.: $b \in {}^\bullet z$ holds, otherwise there exists another witness $e < z$. So there exists a event u such that $u \leq y$ with $b \in {}^\bullet u$ and $z \neq u$.

We now show that $\lceil x \rceil \cup \lfloor u \rfloor \cup \lfloor z \rfloor$, for short $\mathcal{C}^{xuz}$, is a configuration. If $\mathcal{C}^{xuz}$ is no configuration, there exists events $k, l \in \mathcal{C}^{xuz}$, with $k \neq l$ so that ${}^\bullet k \cap {}^\bullet l \neq \varnothing$. For two events of a single configuration ($\lceil x \rceil$, $\lfloor u \rfloor$ and $\lfloor z \rfloor$), no such events exist. So one of the following holds:

- $k \in \lceil x \rceil$, $l \in \lfloor z \rfloor$. As $k \leq x$, $l < z$ and $\neg(x \# z)$, it follows that ${}^\bullet k \cap {}^\bullet l = \varnothing$.
- $k \in \lceil x \rceil$, $l \in \lfloor u \rfloor$. As $k \leq x$, $l < u \leq y$ and, by assumption, $\neg(x \# y)$, it follows that ${}^\bullet k \cap {}^\bullet l = \varnothing$.
- $k \in \lfloor u \rfloor$, $l \in \lfloor z \rfloor$. We chose the witness z to be minimal, so for all events $e < z$ it holds that $\neg(e \# y)$. As $k < u \leq y$ and $l \leq e < z$, it follows that ${}^\bullet k \cap {}^\bullet l = \varnothing$.

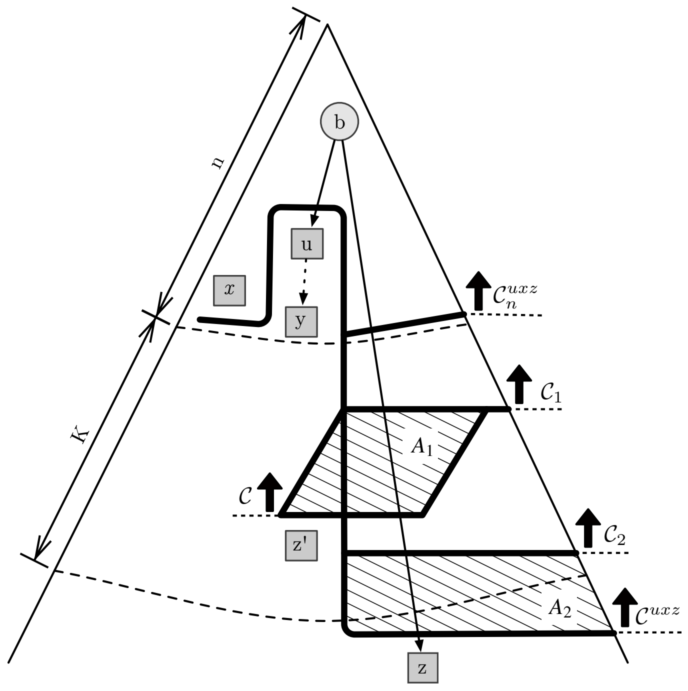


Figure 5.7: Rough sketch of the proof of Theorem 2; there exists a condition b in the preset of both u and z; moreover, $u < y$ and $n = max(\mathcal{H}(x), \mathcal{H}(y))$. From $\mathcal{C}^{uxz}$ we construct the smaller configuration $\mathcal{C}$.

So $\mathcal{C}^{xuz}$ is a configuration. Denote with $\mathcal{C}_n^{xuz} \subseteq \mathcal{C}^{xuz}$ the configuration that contains all events of $\mathcal{C}^{xuz}$ up to height n. Further, assume $\mathcal{H}(z) > \kappa + n$. The height of z and Lemma 11 implies that we have two marking equivalent configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ with $\mathcal{C}_n^{xuz} \subseteq \mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq \mathcal{C}^{xuz}$ and $\mathcal{H}(C_1) < \mathcal{H}(C_2)$. From that it follows that there exists a set of events A_2 so that $\mathcal{C}_2 \overset{A_2}{\rightsquigarrow} \mathcal{C}^{xuz}$ holds. As $M(\mathcal{C}_1) = M(\mathcal{C}_2)$, there exists a set of events A_1 with $f(A_2) = f(A_1)$, so that $\mathcal{C}_1 \overset{A_1}{\rightsquigarrow}$ holds, and with that there exists a configuration $\mathcal{C}$ so that $\mathcal{C}_1 \overset{A_1}{\rightsquigarrow} \mathcal{C}$ holds. Moreover, there exists some event $z' \in A_1$, so that $f(z') = f(z)$ and $\mathcal{H}(z') < \mathcal{H}(z)$.

We now show that $\mathbf{wit}(\neg(x \rhd y), z')$ holds, i.e. z' is also a witness, with strictly smaller height. As $\mathcal{C}$ contains x and $\mathcal{C} \overset{z'}{\rightsquigarrow}$ holds, it follows that $\neg(z' \# x)$. It remains to show that $z' \# y$ holds. As A_1 and A_2 are isomorphic, A_2 contains a condition $b' \in {}^\bullet z'$ with $f(b) = f(b')$.

Conditions b and b' are both contained in $\mathcal{C}$ so we apply Lemma 13, and with that, one of the following statements holds: $b = b'$, $b < b'$ or $b > b'$.

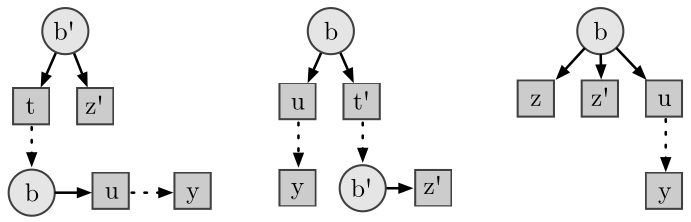


Figure 5.8: Possible situations, as $b' \rightleftharpoons b$.

We show that in each case we obtain $z' \# y$. We additionally need following Lemma for that.

Lemma 14 (Source transitions of events z and u are different)**.** *$f(z) \neq f(u)$ holds.*

Proof. Assume $f(z) = f(u)$ holds. There must exist two conditions c and c' in the presets of z and u, respectively, with $\mathcal{H}(c') < \mathcal{H}(c)$ and $f(c) = f(c')$. This is because $\mathcal{H}(z) > n$ and $\mathcal{H}(u) < n$, so there exists a condition c in the preset of z with $\mathcal{H}(c) = \mathcal{H}(z) - 1 \geq n$. As, by assumption $f(z) = f(u)$, there exists a condition c' in the preset of u with $f(c') = f(c)$ and $\mathcal{H}(c') < \mathcal{H}(n)$. So $\mathcal{H}(c') < \mathcal{H}(c)$. Conditions c and c' are in causal relation, as they cannot be in co-relation or conflict, because both are contained in the same configuration $\mathcal{C}^{xuz}$ and are copies of the same place. Together with the height constraint, we have $c' < c$. With that, there must be some new event t, that consumes c' with hight strictly lesser than z. But then, this t would be a new reveals witness, because it is in conflict with y, but not with x. This contradicts the fact the z is the smallest reveals witness. $\square$

First we show that $u \notin \mathcal{C}$. Assume the contrary, $u \in \mathcal{C}$. Then $u \in A_1$, as $u \notin \mathcal{C}_1$ holds because of $u \notin \mathcal{C}^{xuz}$ and $\mathcal{C}_1 \subseteq \mathcal{C}^{xuz}$. Because of the isomorphism between A_1 and A_2 we have an event $u'' \in A_2$ with $f(u'') = f(u)$. Conditions $b'' \in {}^{\bullet}u''$ and $b \in {}^{\bullet}u$ with $f(b'') = f(b)$ are contained in $\mathcal{C}^{xuz}$ as $\mathcal{C}^{xuz} \overset{u}{\rightsquigarrow}$ and $u'' \in A_2$ hold. So Lemma 13 implies that b'' and b are in causal relation and, as $u \notin \mathcal{C}^{xuz}$ but $u'' \in \mathcal{C}^{xuz}$, we immediately get $u \# u''$ for $b'' < b, b'' > b$ and $b'' = b$. On the other hand, we get $\neg(u \# u'')$ as $u'' \in \mathcal{C}^{xuz}$ and $\mathcal{C}^{xuz} \overset{u}{\rightsquigarrow}$. Both cannot hold so we obtain $u \notin \mathcal{C}$. Using this result we now show that $z' \# y$. Figure 5.8 shows the cases we have to consider.

- Assume $b > b'$ holds (see Figure 5.8, left). Then there exists an event

t, so that $b' \in {}^{\bullet}t$ and $t < b$. Assume $z' = t$. Then $z' < b$ but then $z' \in \mathcal{C}_n^{xuz}$, which is not the case. So $z' \neq t$ holds and with that $z' \# y$.

- Assume $b < b'$ holds (see Figure 5.8, middle). Then there exists an event t', so that $t' < b'$ and $b \in {}^{\bullet}t'$. As $t' \in \mathcal{C}$ (because $t' < z'$) and $u \notin \mathcal{C}$ as proven before we have $t' \neq u$ with $b \in {}^{\bullet}t' \cap {}^{\bullet}u$ so $z' \# y$.

- Assume $b = b'$ (see Figure 5.8, right). Lemma 14 denotes that $f(z) \neq f(u)$. As $f(z) = f(z')$, we have $f(z') \neq f(u)$ and so $z' \neq u$ holds. By definition $b \in {}^{\bullet}z' \cap {}^{\bullet}u$, so $z' \# y$.

In summery, for each event z with $\mathbf{wit}(\neg(x \triangleright y), z)$ and $\mathcal{H}(z) > n + \kappa$, there exists another event z', so that $\mathbf{wit}(\neg(x \triangleright y), z')$ holds with $\mathcal{H}(z') < \mathcal{H}(z)$. If z' still has greater height than $n + \kappa$ we iterate the given procedure until we find a witness with height less or equal than $n + \kappa$. □

5.4 Computing The Reveals Relation

Section 5.2 gives decidability results for the reveals relation, i.e. deciding $a \triangleright b$ can be done by analysing the finite unfolding prefix of height $max(\mathcal{H}(a), \mathcal{H}(b)) + \kappa$ where κ is a constant depending on the net. In this section we take this result and, on that foundation, present algorithms for (1) explicitly computing the reveals relation for two events and (2) deciding the reveals relation property for all possible pairs of a set of events. Moreover we report on an efficient implementation and experiments.

5.4.1 Algorithms

As already mentioned we first propose an algorithm for computing the reveals property for one concrete relation pair, i.e. deciding whether $a \triangleright b$ for events a and b. The algorithm is derived straight forward from the presented decidability result and we believe that there is not much optimization potential for solving this concrete problem. This is in strong contrast to deciding the reveals property for a large set pairs of events. We derive an algorithm that reduces the costs for the computation of the reveals property for individual pairs of events by deciding that property on a larger set of events at once.

This is done by exploiting the fact that the relation property can be, in defined limits, inherited from other events and expressed in terms of different relations on nets that are efficiently computable. This makes also sense in terms of diagnosis. Typically, the information "$a \triangleright b$" is not so valuable as the information "a reveals all events in set A and is revealed by all events in set B", so it makes sense to consider computing the reveals property between many events at once, especially if both information can be gained with almost the same effort.

Single Pair Reveals Computation

From the proofs presented in Section 5.2 we derive a straight forward algorithm for deciding $x \triangleright y$ on events x and y contained in unfolding U. Note that the idea for this algorithm goes back to Stefan Schwoon. Assume that we already computed a prefix of U containing x and y. Check if $\neg(x\#y)$ holds and if the answer is positive return $\neg(x \triangleright y)$ and we are done. If $\neg(x \triangleright y)$ holds, there exists event z with $\mathcal{H}(z) \leq n + \kappa$ and $\mathbf{wit}(\neg(x \triangleright y), z)$ where n is the maximal height of x and y. So we have to find an event z such that $\neg(z \# x)$ and $z \# y$. From $z \# y$ it follows that there exists condition b with $b < y$ such that $b \in {}^{\bullet}z$. We mark all such conditions b as "targets". Further we know that events z' with $\neg(z' < y)$ or $\neg(z' \geq y)$ are no candidates for z as they are not in conflict with it. We mark these events and its post-conditions as "dead". We now play the following game: find an event z so that ${}^{\bullet}z$ contains a condition marked as target such that $\neg(z\#x)$. So we check each event that is not dead in the prefix for this condition and if we find one we return that $\neg(x \triangleright y)$ holds. Otherwise, we unfold the prefix further not extending the prefix with possible extensions of dead conditions and possible extensions with height greater then $n + \kappa$. This unfolding procedure necessarily terminates due to the height constraint. On adding a new event we check if its preset contains a marked target condition and if $\neg(z\#x)$ holds. If both is satisfied, we report that $\neg(x \triangleright y)$ holds. If the procedure terminates without finding such an event, we return that $x \triangleright y$ holds.

Computing Reveals for Prefix

In this section we present an algorithm that, given a finite occurrence net $O = (C, E, F_O, C_0)$, computes for all pairs $(x, y) \in E \times E$ the reveals relation property, i.e. determines whether $x \triangleright y$ holds. For short, we say we compute reveals on the set of events E.

Typically, we consider a net $N = (P, T, F_N, M_0)$ with its respective unfolding $U = (C', E', F_U, C'_0)$, that is an occurrence net as defined in Section 3.2.3. We want to compute reveals on E'. In most cases, U is infinite and so we cannot apply our algorithm to this net but to a finite prefix of it, i.e. to an occurrence net $P = (C'', E'', F_P, C''_0)$ with $P \sqsubseteq U$.

Assume $\neg(x \rhd y)$. Then there exists an event z so that $\mathbf{wit}(\neg(x \rhd y), z)$ holds. As $P \sqsubseteq U$, it is possible that although $\neg(x \rhd y)$ holds, no witness for this fact is contained in P, i.e. for all z so that $\mathbf{wit}(\neg(x \rhd y), z)$, $z \notin E''$. So our algorithm applied to a finite prefix of U might return the spurious result that $x \rhd y$ holds. On the other hand, if our algorithm returns $\neg(x \rhd y)$ for two events x and y then this answer is always correct.

Moreover, Theorem 2 denotes that the answer for "does $x \rhd y$ hold" of our algorithm on P is correct for pairs $(x, y) \in E \times E$ if the height of prefix P is greater or equal than $max(\mathcal{H}(x), \mathcal{H}(y)) + \kappa$. So if we are interested in deciding reveals on the set of events H then we generate the maximal prefix with height $\mathcal{H}(H) + \kappa$ and run our algorithm on this prefix.

Now we discuss our algorithm in detail. We first introduce to ways to iterate over all events of a finite occurrence net $O = (C, E, F_O, C_0)$ – *bottom-up* and *top-down*. These iteration techniques ensure that each event is visited exactly once. Moreover it is assumed that all $<$-maximal elements in O are conditions.

- *Bottom-up* – An event is visited the first time if all its causal succeeding events have been visited.
  We store a counter for each event and condition, initialized to its number of post-conditions and post-events, respectively. Additionally we maintain a working queue of events initialized empty. At the beginning we iterate over the $<$-maximal conditions (conditions with counter 0). We decrease the counter of each pre-event (if existent) and if such a counter becomes 0 we add this event to the queue. Now, while not empty, we iterate the following:

  We pull an event from the queue and visit it. Then we iterate over the pre-conditions of it that themselves have a pre-event. We decrease the counter of the pre-condition and if the counter of this pre-condition becomes 0 we additionally decrease the counter of the pre-event. If the counter of this pre-event becomes 0 we add it to the queue.

- *Top-down* – An event is visited the first time if all its causal preceding events have been visited.

  The idea here is very similar to the above one. We store a counter for each event, initialized with the amount of its pre-conditions. We also maintain a working queue of conditions, initialized with the $<$-minimal conditions (conditions without predecessors). While not empty we iterate the following:

  We pull a condition from the queue, and, for each post-event, we decrease the counter of this event. If the counter of an event becomes 0 we visit that event and add all its post-conditions to the working queue.

We now discuss the complexity of these iteration algorithms. Considering the bottom-up algorithm, each event or condition is not added more than once to the queue and in each step we remove one condition from the queue, so we have at most $|E|$ steps. In each such step we iterate over pre-conditions and for each pre-condition we (1) decrement one (or two) counter(s) and (2) possibly add an event to the queue. Assume (1) and (2) to be atomic. An event can have at most $|C|$ pre-conditions. Let $n = |C| + |E|$. Then we have a worst case running time of $\mathcal{O}(n^2)$. Now consider the top-down iteration. Each condition is added at most once to the queue and in each step we remove one from it, so we have at most $|C|$ steps. In each such step we iterate over post-events and possibly over their post-conditions. That are at most $|C| + |E|$ elements. For each such element we (1) decrement a counter and (2) possibly add a condition to the queue. Assume (1) and (2) to be atomic. Analogous to the case above, we have a worst case running time of $\mathcal{O}(n^2)$.

This is the complexity result for general occurrence nets $O = (C, E, F_O, C_0)$. In our work we consider finite occurrence nets (prefixes) that are created by unfoldings of nets $N = (P, T, F_N, C_0)$. But then we have a new bound on the number of post/pre-conditions, i.e. $|T|$, a constant. With that, for the bottom-up algorithm we now have $\mathcal{O}(n)$ worst case running time. Although we cannot bound the amount of post-events of some condition in such a way, a huge branching factor on conditions in a practical unfolding is very unlikely so we have average running time of $\mathcal{O}(n)$ in this case.

Now we present the actual reveals computation. Relations are represented as arrays of sets. For that assume relation R. For each pair aRb, the set $R[a]$ contains element b. We compute the reveals relation in three passes over the prefix establishing the $\leq$-relation ($a \in \leq [b]$, iff $b \leq a$), conflict-relation and finally the reveals-relation.

The $\leq$-relation is established in a bottom-up iteration over the set of events E. Algorithm 8 shows the computation of $\leq [e]$ for single event e. The iteration guaranties that when visiting an event, the sets ($\leq [e']$) for all events $e' > e$ have already been computed. These sets are passed as partial causal relation $\leq []$ to the algorithm. We exploit the fact that the set of causal successors of an event is the union of the causal successors of each event. We return the array $\leq []$ that now additionally contains the completed set $\leq [e]$.

Algorithm 8: Compute causal set for event e

```
input  : Event $e$, partial causal relation $\leq []$
output: Set of events $\leq [e]$ in causal relation with event $e$
begin
    $\leq [e] \longleftarrow \{e\}$;
    foreach $e' \in (e^\bullet)^\bullet$ do
        $\leq [e] \longleftarrow (\leq [e]) \bigcup (\leq [e'])$;
    return $\leq []$;
```

The conflict-relation is established in a top-down iteration over the set of events E. We exploit the fact that event e inherits the conflicts of its preceding events and is additionally in conflict with all causal successors of its pre-conditions, except itself, and nothing else.

Algorithm 9 shows the computation of $\#[e]$ for single event e. We already have computed the causal successors for each event in Algorithm 8, passed as array $\leq []$ to the algorithm. Moreover, because of the top-down iteration, the conflict sets of all preceding events have already been computed and are passed as partial conflict relation $\#[]$. The array $\#[]$ is returned, additionally containing completed set $\#[e]$.

In order to compute the reveals relation we exploit following facts:

- If $x \# y$ holds for two events x and y then event x does not reveal event y. (Let $z = x$. Then we have $\mathbf{wit}(\neg(x \triangleright y), z)$).

Algorithm 9: Compute conflict set for event e

```
input  : Event $e$, causal relation $\leq []$, partial conflict relation $\#[]$
output: Conflict set $\#[e]$ for event $e$
begin
    $\#[e] \longleftarrow \varnothing$;
    foreach $e' \in {}^\bullet({}^\bullet e)$ do
        $\#[e] \longleftarrow (\#[e]) \bigcup (\#[e'])$;
    foreach $c \in ({}^\bullet e)$ do
        foreach $e' \in (c^\bullet \setminus e)$ do
            $\#[e] \longleftarrow (\#[e]) \bigcup (\leq [e'])$;
    return $\#[]$;
```

- Event x reveals all events revealed by its causal predecessors. This is because $\#y \subseteq \#x$ for all $y < x$.

Similar to the conflict-relation the reveals-relation is computed in a top-down iteration over the set of events E. So when computing the reveals set for event e we have already computed $\triangleright[e']$ for all preceding events $e' < e$. Algorithm 10 shows the computation of $\triangleright[e]$ for single event e. For each such event e we exploit the two mentioned facts. Fact two implies that each event revealed by an event e' in ${}^\bullet({}^\bullet e')$ is also revealed by e. These already computed reveal sets are passed as partial reveals relation $\triangleright[]$ and added in the first "foreach" loop to $\triangleright[e]$. Additionally, we know that no event in conflict with e is revealed by it. We already have computed the conflict set of e with Algorithm 9 that is passed as array $\#[]$. So we have a set of events revealed by e and another set of events not revealed by e. It remains a set of events where we do not know whether or not a contained event is revealed by e. In the second "foreach" loop we iterate over these remaining events and explicitly check the reveals property. It is obvious that this algorithm is correct. The array $\triangleright[]$ is returned, additionally containing the completed set $\triangleright[e]$.

Algorithm 11 summarizes the computation of reveals in one procedure. For the following complexity consideration, assume again $n = |C| + |E|$. We use in worst case space of $\mathcal{O}(3n^2) = \mathcal{O}(n^2)$ as storing each of the three

Algorithm 10: Compute reveals set for event e

```
input  : Event $e$, conflict relation $\#[]$, partial reveals relation $\triangleright[]$
output: Reveals set $\triangleright[e]$ for event $e$
begin
    $\triangleright[e] \longleftarrow \{e\}$;
    foreach $e' \in {}^\bullet({}^\bullet e)$ do
        $\triangleright[e] \longleftarrow (\triangleright[e]) \bigcup (\triangleright[e'])$;
    foreach $e' \in (E \setminus (\triangleright[e] \bigcup \#[e]))$ do
        if $\#[e] \subseteq \#[e']$ then
            $\triangleright[e] \longleftarrow (\triangleright[e]) \cup e'$;
    return $\triangleright[]$;
```

Algorithm 11: Compute reveals

```
input  : Occurrence Net $O = (C, E, F_O, C_0)$
output: Reveals relation $\triangleright[]$
begin
    foreach (bottom-up iteration) $e \in E$ do
        $(\leq [])$ ← Algorithm 8 executed on input $(e, \leq [])$
    foreach (top-down iteration) $e \in E$ do
        $(\#[])$ ← Algorithm 9 executed input $(e, \leq [], \#[])$
    foreach (top-down iteration) $e \in E$ do
        $(\triangleright[])$ ← Algorithm 10 executed input $(e, \#[], \triangleright[])$
    return $\triangleright[]$;
```

relations takes quadratic space in amount of events. Storing the queues, as well as the set of events that are explicitly checked for reveals consumes strictly less than quadratic space, so this is not considered here.

We iterate 3 times over O, each time iterating over k events. The iteration has worst case running time of $\mathcal{O}(n^2)$ and average case running time of $\mathcal{O}(n)$, as already discussed before. It is easy too see that following facts hold (a set of events is always smaller than $|E|$):

$$|\{e' \mid e' \in (e^\bullet)^\bullet\}| \leq |E| \text{ for all } e \in E$$
$$|\{e' \mid e' \in {}^\bullet({}^\bullet e)\}| \leq |E| \text{ for all } e \in E$$
$$|\{e' \mid e' \in (c^\bullet \setminus e), c \in ({}^\bullet e)\}| \leq |E| \text{ for all } e \in E$$
$$|\{e' \mid e' \in (E \setminus (\triangleright[e] \bigcup \#[e]))\}| \leq |E| \text{ for all } e \in E$$

Assume that set operations are atomic. Then, the processing of one single event (shown in Algorithms 8 - 10) has worst case running time of $\mathcal{O}(n)$. Together with the complexity result for the iteration algorithm, we have worst case running time of $\mathcal{O}(n^3)$ and average running time of $\mathcal{O}(n^2)$ for computing reveals on a finite prefix of some petri net.

So summarized, on average the complexity of computing reveals on a finite prefix is quadratic in both, time and space.

5.4.2 Experiments

In this section we give experimental data on the height and computation time of the level-2-unfoldings of middle-sized Petri-net benchmark instances. Then we present an efficient implementation of the presented algorithm for computing reveals on a given prefix and report on results.

The Level-2-Unfolding

Table 5.1 shows experiments on middle-sized safe Petri nets, taken from the PEP framework. These instances are considered standard benchmark instances in the Petri net community.

Name number of places and transitions is shown in the columns "Petri net", $|P|$ and $|T|$, respective. The Petri net unfolder mole is modified to construct the level-2-unfolding of the respective net, and "Time[s]" shows the amount of time needed for this construction. The column with caption "κ" shows the height of the respective "level-2-unfolding". The columns L_1[e] and $L_2[e]$ shows the size of the "level-1-unfolding" and "level-2-unfolding" in number of events, respectively. Remember that "$\kappa + n$" is the height up to which we have to unfold events, to decide reveals on events with height n. The experiments show that the construction of this prefix is usually done in a few seconds (except for the net “mutual”). This shows that a prefix for deciding reveals on is practically computable. Measured in events, the “level-2-unfolding” has on average ten times the size of the “level-1-unfolding”. The values for κ are also very promising since it is roughly twice the number of transitions of the net.

Table 5.1: Net statistics and computation of K

Petri net	$\|P\|$	$\|T\|$	κ	Time[s]	L_1[e]	L_2[e]
buf100	200	101	201	1.6	5051	5152
elevator	59	74	61	0.1	293	3798
gas_station	30	18	19	0.1	20	30
mutual	62	67	109	19.1	497	403807
parrow	77	54	72	0.2	284	5402
peterson	27	31	26	0.1	49	621
reader_writer_2	53	60	26	0.4	147	18250
sdl_arq	208	234	114	0.1	199	1937
sdl_arq_deadlock	202	183	40	0.1	41	67
sdl_example	323	471	71	0.1	132	132
sem	26	25	18	0.1	32	92

Experiments on Reveals Unfolding

We implemented the algorithms presented in Section 5.4.1. The implementation was done in Java, relying heavily on bitset operations.

A single bitset encapsulates a vector of bits. A binary relation $R \subseteq A \times A$ is stored as a list of $|A|$ bitsets with length $|A|$. Each element $a \in A$ is associated with a unique natural number $g(a) \in [1, |A|]$, analogous for b. If $(a, b) \in R$, we set the bit at position $g(b)$ in bitset vector at list position $g(a)$.

Representing all relations as lists of bitsets, almost all operations necessary for computing reveals (and the other relations) are expressible as logical operations like OR and XOR between bitsets. As these operations are implemented in hardware on the CPU, the whole computation is very fast.

We benchmarked our implementation on unfolding prefixes of sizes from thousands to tens of thousands events. As the complete prefixes of the considered nets are very small, bigger prefixes are generated. The results are shown in Table 5.2. Name of the net and the size of its unfolding in events is shown in the columns "Petri net" and "Events", respectively. For each such prefix, we measured the computation time (in seconds) of our implementation, split into the three passes of the algorithm – $\leq$-relation, conflict-relation and reveals-relation, shown in columns "post", "conf" and "ref", respectively.

The implementation scales very well, with computing times ranging from a few seconds to minutes, depending on the size of the unfolding. Usually, computing the $\leq$-relation takes less time than computing the conflict-relation and computing the reveals relation takes most of the time. Nevertheless, all of three passes are of the same order of magnitude. The experiments altogether demonstrate that we are able to compute the reveals relation practically for many events at once.

5.5 Conclusion

The reveals relation contributes to net diagnosis, by allowing reasoning about invisible events and abstracting occurrence nets. The reveals problem "$a \rhd b$?" was proven to be decidable in (infinite) occurrence nets before. Nevertheless the derived algorithms have very high complexity, rendering practical computation infeasible.

We contributed to solving the reveals problem in two ways:

- *Theoretically:* we notably improved the existing decidability result, lifting the reveals problem from decidable to practical computable. This is confirmed with experimental data on standard Petri net benchmark instances.

- *Practically:* we showed an efficient algorithm and implementation for deciding the reveals problem. We report on experiments, where we actually compute the reveals relation between all events in occurrence nets with ten of thousands of events.

Table 5.2: Running times of computing reveals

Petri net	Events	post (Time/s)	conf (Time/s)	rev (Time/s)
bds_1.sync	12900	0.13	0.19	0.30
buf100	17700	0.17	0.12	0.25
byzagr4_1b	14724	0.18	0.19	0.68
dpd_7.sync	10457	0.11	0.15	0.24
dph_7.dlmcs	37272	0.56	0.91	2.10
elevator75	234879	15.84	22.58	97.47
elevator	5586	0.05	0.05	0.13
elevator_4	16856	0.17	0.27	0.38
fifo20	100696	2.92	3.72	22.88
ftp_1.sync	83889	2.08	3.61	6.78
furnace_3	25394	0.29	0.47	0.95
gas_station	2861	0.01	0.01	0.01
key_4.fsa	67954	1.40	2.19	4.62
parrow	85869	2.47	4.17	9.51
peterson	72829	1.60	2.54	5.23
q_1.sync	10722	0.11	0.15	0.30
q_1	7469	0.08	0.09	0.17
reader_writer_2	20229	0.24	0.37	0.53
rw_12.sync	98361	2.36	5.14	6.36
rw_12	49179	0.68	1.25	1.70
rw_1w3r	15401	0.15	0.22	0.50
rw_2w1r	9241	0.10	0.11	0.25
sdl_arq	2691	0.03	0.03	0.09
sem	19689	0.20	0.23	0.61

Chapter 6

Conclusion

In this chapter we discuss to which extend our objectives were fulfilled, i.e. we summarize the presented work, discuss its weaknesses and present future work. This thesis revolves around the question that we asked at its beginning: "to which extend can the task of debugging be automated?". With debugging, we not only refer to the task of fixing a known defect of some program but also to the task of its detection. We therefore consider the bigger cyclic activity of monitoring a system for showing a potential failure, analysing the system to check if the error is spurious and if not, triggering the automatic correction, restarting the system and iterating this procedure. This approach is shown in Figure 1.1. However, we do not present a holistic approach, but focus on the one hand on diagnosis and on the other hand on automatic error correction. Implementing the "glue" between these two approaches would very likely yield another thesis.

Usually, techniques and tools originating from the research area of model checking are not very industrial relevant. There are some positive counterexamples like, e.g. the Astrée Static Analyzer [8], a model checker that was successfully used to verify tens of thousands of lines of code in the flight controller software of the Airbus company or SAT and BDD based methods used for chip verification, e.g. at the Intel company [46]. In mainstream development however, when we look at static analysis tools that are heavily used, we identify mainly lightweight tools like, e.g. the Google CodePro Analytix Tool[1], helping the programmer to write more clean code or enforcing basic code coverage.

[1] `https://developers.google.com/java-dev-tools/codepro/doc/`

The approaches in this research area always get more sophisticated, from a theoretical point of view, but even the older tools are usually not matured or are extremely highly specialised. For the error correction part approach, we therefore decided to base on a matured tool not utilizing some sophisticated model of the software system. This was the reason for using Java Path Finder as base for our implementation, using bare Java virtual machine snapshots as model state representation. Applying search and testing techniques we could automatically fix real-world code. Still there are some drawbacks of this approach and there is additionally space for future work. We require that the specification is complete. Writing a complete specification is a very time consuming task and so it is, in most cases, an unrealistic assumption that the programmer will take the time to give a complete specification when trying to fix a bug. However, correcting an error fully automatically without a specification is not possible, so a semi-automatic approach might be better suited. Moreover, in our case study, we use a restricted catalogue of error prone code together with alternatives. Still, with this restricted catalogue we could fix real-world code, but it is up to future work to make experiments with a more sophisticated catalogue, e.g. handling complex data structures like linked lists.

For the diagnosis part of this work, we focus on distributed systems that are partially observable. The output of this research is not a tool for code analysis, but theoretical results and their implementations together with practical verifications using case studies. We present two approaches, one supporting diagnosis and one applying diagnosis in Petri net systems.

For supporting diagnosis, we present the computation of the reveals relation in Petri net unfoldings. This relation is stated as follows: "a reveals b iff whenever a occurs b will eventually occur or has already occurred". We consider diagnosis of partial observable systems, so when a is visible and b not, it is clearly of advantage to have this non-trivial relation at ones disposal. This relation was introduced and shown to be decidable by Stefan Haar [3]. However, crucial for its appliance is the question how "fast" it can be decided. The complexity bound for deciding it, found in [3], is very pessimistic. Loosely speaking and simplified, for deciding "a reveals b", we have to additional "unfold" the prefix that contains a and b n times, with n, the number of reachable markings of the net and then we have to do computations on this large prefix. Clearly, this cannot be implemented efficiently. We notably reduce the complexity bound, i.e. the parameter n becomes the constant 2.

Then we present an efficient, bitset based algorithm, to actually decide the relation on a set of events with almost only using bitset operations. Before this work, it was not clear if a tractable algorithm for solving this problem exists at all. We showed its existence, designed and implemented it and presented benchmarks with it. This gives hope that this algorithm actually can be used in practice. To show a practical application of the algorithm is, however, up to future work.

For applying diagnosis, we consider the problem of, given the observation of a partial observable system and a system alarm, deciding if the alarm is spurious and deriving an explanation for a nonspurious alarm. Basis for the explanation are the behaviours of the system leading to this alarm, represented as unfolding prefix. Generating this prefix is the fundamental task of our approach. Our aim was not to generate the prefix when the observation is complete, as this problem has already been solved by computing the product of observation and system and unfolding that. Instead, we wanted to design an ad-hoc unfolding algorithm, that can run in parallel to the system and even speculates about the systems future. This approach has two fundamental benefits: (1) when an alarm is triggered, we are already (almost) finished with the unfolding and (2) when looking into the systems future (proactive diagnosis), we might detect possible alarms in the systems future. As the problem of keeping the unfolding synchronous with the observation is computationally complex, we introduce different levels of overapproximations for the unfolding prefix that are more easily to compute. This adds behaviours to the unfolding that might not be compatible with the observation. To compensate that, we do not analyze the unfolding prefix directly but use sat solving methods to "filter" the unfolding prefix and assure that we only consider valid net behaviours. When doing proactive diagnosis, we additionally have to deal with problem of possibly adding "garbage" to the unfolding, i.e. future system behaviour we speculated about that is falsified later. We present an efficient data structure for allowing fast removal of this garbage. In an extensive case study we benchmark our implementation and justify the mentioned overapproximation of the unfolding. We experimentally showed that the overapproximation behaves very well with the growing observation and contains almost only valid behaviours. We also benchmarked the speed improvement we get when doing the diagnosis proactively. There is, in most cases, a small speed benefit. However, this benefit is not outstanding. As already mentioned, proactive diagnosis additionally allows detecting possible alarms in the systems future, however, this was not explored in a case study and is still up to future work.

In summary, we presented three approaches, one for program repair and two for diagnosis. Each approach is not pure theory, but implemented and justified using extensive case studies. Therefore, with our presented approaches, we could contribute significantly and in various ways to the model checking community.

Bibliography

[1] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, pages 97–105, 2003. (cited on p 8)

[2] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001. (cited on p 8)

[3] Albert Benveniste, Eric Fabre, Stefan Haar, and Claude Jard. Diagnosis of asynchronous discrete-event systems: a net unfolding approach. *IEEE Transactions on Automatic Control*, 48(5):714–727, 2003. (cited on pp 31, 41, 55, 82, 108)

[4] Allan Cheng, Javier Esparza, Jens Palsberg, and Jens Palsberg. Complexity results for 1-safe nets. pages 326–337, 1995. (cited on pp 34, 60)

[5] III Cikanek, H. Space shuttle main engine failure detection. *Control Systems Magazine, IEEE*, 6(3):13 –18, june 1986. (cited on p 29)

[6] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. (cited on p 8)

[7] Holger Cleve and Andreas Zeller. Locating causes of program failures. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 342–351. ACM, 2005. (cited on p 8)

[8] P. Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1-2):3, 2000. (cited on p 107)

[9] J.M. Couvreur, D. Poitrenaud, and P. Weil. Branching processes of general petri nets. *Applications and Theory of Petri Nets*, pages 129–148, 2011. (cited on p 38)

[10] J. Culberson. Sokoban is pspace-complete. In *Fun With Algorithms*, volume 4, pages 65–76. Citeseer, 1999. (cited on p 75)

[11] V. Diekert and G. Rozenberg. *The book of traces.* World Scientific Pub Co Inc, 1995. (cited on pp 61, 67, 70)

[12] N. Eén and N. Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 333–336. Springer, 2004. (cited on p 79)

[13] Joost Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28(6):575–591, 1991. (cited on pp 35, 39, 40)

[14] Javier Esparza and Christian Kern. Reactive and proactive diagnosis of distributed systems using net unfoldings. In *ACSD*, 2012. To appear. (cited on p 5)

[15] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002. (cited on pp 35, 39, 40, 66, 67)

[16] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002. (cited on p 44)

[17] Eric Fabre and Albert Benveniste. Partial order techniques for distributed discrete event systems: Why you cannot avoid using them. *Discrete Event Dynamic Systems*, 17(3):355–403, 2007. (cited on pp 31, 41)

[18] Eric Fabre, Albert Benveniste, Stefan Haar, and Claude Jard. Distributed monitoring of concurrent and asynchronous systems. *Discrete Event Dynamic Systems*, 15(1):33–84, 2005. (cited on pp 31, 41)

[19] S. Genc and S. Lafortune. Distributed diagnosis of discrete-event systems using petri nets. *Applications and Theory of Petri Nets 2003*, pages 316–336, 2003. (cited on p 31)

[20] A. Grastien, J.R. Anbulagan, and E. Kelareva. Modeling and solving diagnosis of discrete-event systems via satisfiability. In *Proc. of the 18th International Workshop on Principles of Diagnosis (DX)*, 2007. (cited on p 31)

[21] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. *Electr. Notes Theor. Comput. Sci.*, 174(4):95–111, 2007. (cited on p 8)

[22] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006. (cited on p 8)

[23] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2003. (cited on p 8)

[24] S. Haar. Types of asynchronous diagnosability and the reveals-relation in occurrence nets. 2009. (cited on pp 84, 86, 88, 89)

[25] Stefan Haar, Christian Kern, and Stefan Schwoon. Computing the reveals relation in occurrence nets. In Giovanna D'Agostino and Salvatore La Torre, editors, *GandALF*, volume 54 of *EPTCS*, pages 31–44, 2011. (cited on p 5)

[26] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002. (cited on p 7)

[27] G.M. Hopper. The first bug. *Annals of the History of Computing*, 3(3):285–286, 1981. (cited on p 1)

[28] R. Isermann. *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Verlag, 2006. (cited on p 27)

[29] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2005. (cited on p 9)

[30] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE*, pages 273–282. ACM, 2005. (cited on p 8)

[31] C.T. Kemper, S. Lowenfeld, and M.S. Fox. Rule based diagnostic system with dynamic alteration capability, February 10 1987. US Patent 4,642,782. (cited on p 28)

[32] Christian Kern and Javier Esparza. Automatic error correction of java programs. In *FMICS*, pages 67–81, 2010. (cited on p 5)

[33] K. Kim and A.G. Parlos. Induction motor fault diagnosis based on neuropredictors and wavelet signal processing. *Mechatronics, IEEE/ASME Transactions on*, 7(2):201–219, 2002. (cited on p 29)

[34] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM. (cited on p 7)

[35] Feng Lin. Diagnosability of discrete event systems and its applications. *Discrete Event Dynamic Systems*, 4:197–212, 1994. 10.1007/BF01441211. (cited on p 31)

[36] Agnes Madalinski, Farid Nouioua, and Philippe Dague. Diagnosability verification with petri net unfoldings. *KES Journal*, 14(2):49–55, 2010. (cited on p 31)

[37] S.A. McIlraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *PRINCIPLES OF KNOWLEDGE REPRESENTATION AND REASONING-INTERNATIONAL CONFERENCE-*, pages 167–179. MORGAN KAUFMANN PUBLISHERS, 1998. (cited on p 31)

[38] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. CAV*, LNCS 663, pages 164–177. Springer, 1992. (cited on pp 40, 44)

[39] R. Milner. Calculi for synchrony and asynchrony. *Theoretical computer science*, 25(3):267–310, 1983. (cited on p 53)

[40] V. Palade and C.D. Bocaniala. *Computational intelligence in fault diagnosis*. Springer Publishing Company, Incorporated, 2010. (cited on pp 27, 28)

[41] D. Park. Concurrency and automata on infinite sequences. *Theoretical computer science*, pages 167–183, 1981. (cited on p 53)

[42] R.J. Patton. Fault detection and diagnosis in aerospace systems using analytical redundancy. *Computing Control Engineering Journal*, 2(3):127 –136, may 1991. (cited on p 28)

[43] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *Automatic Control, IEEE Transactions on*, 40(9):1555–1575, 1995. (cited on p 31)

[44] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *Control Systems Technology, IEEE Transactions on*, 4(2):105–124, 1996. (cited on p 31)

[45] Stefan Schwoon. The *Mole* tool. `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/`. (cited on p 79)

[46] C.J.H. Seger, R.B. Jones, J.W. O'Leary, T. Melham, M.D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(9):1381–1405, 2005. (cited on p 107)

[47] T. Sorsa, H.N. Koivo, and H. Koivisto. Neural networks in process fault diagnosis. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(4):815 –825, jul/aug 1991. (cited on p 30)

[48] Stefan Staber, Barbara Jobstmann, and Roderick Bloem. Finding and fixing faults. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2005. (cited on p 9)

[49] V. Venkatasubramanian and K. Chan. A neural network methodology for process fault diagnosis. *AIChE Journal*, 35(12):1993–2002, 1989. (cited on p 28)

[50] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003. (cited on p 8)

[51] Chao Wang, Zijiang Yang, Franjo Ivancic, and Aarti Gupta. Whodunit? causal analysis for counterexamples. In Susanne Graf and Wenhui Zhang, editors, *ATVA*, volume 4218 of *Lecture Notes in Computer Science*, pages 82–95. Springer, 2006. (cited on p 8)

[52] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374. IEEE, 2009. (cited on p 8)

[53] T.S. Yoo and S. Lafortune. Polynomial-time verification of diagnosability of partially observed discrete-event systems. *Automatic Control, IEEE Transactions on*, 47(9):1491–1495, 2002. (cited on p 31)

[54] A. Zeller. *Why programs fail: a guide to systematic debugging.* Morgan Kaufmann, 2009. (cited on p 7)

[55] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, 2002. (cited on p 7)

[56] Y. Zhang, X. Ding, Y. Liu, and PJ Griffin. An artificial neural network approach to transformer fault diagnosis. *Power Delivery, IEEE Transactions on*, 11(4):1836–1841, 1996. (cited on p 28)

List of Figures

List of Tables

List of Algorithms