The Causal Graph Revisited for Directed Model Checking

Martin Wehrle and Malte Helmert

University of Freiburg, Germany
{mwehrle,helmert}@informatik.uni-freiburg.de

Abstract. Directed model checking is a well-established technique to tackle the state explosion problem when the aim is to find error states in large systems. In this approach, the state space traversal is guided through a function that estimates the distance to nearest error states. States with lower estimates are preferably expanded during the search. Obviously, the challenge is to develop distance functions that are efficiently computable on the one hand and as informative as possible on the other hand. In this paper, we introduce the *causal graph* structure to the context of directed model checking. Based on causal graph analysis, we first adapt a distance estimation function from AI planning to directed model checking. Furthermore, we investigate an abstraction that is guaranteed to preserve error states. The experimental evaluation shows the practical potential of these techniques.

1 Introduction

Directed model checking is a well-established technique to efficiently detect error states in large systems. In this approach, a distance heuristic is used to estimate the distance of each state encountered during the state space traversal to a nearest error state. The search then prefers states with lower estimated error distance. Obviously, the success of this approach crucially depends on the quality of this distance function. On the one hand, it should be as informative as possible to only explore a relatively low number of states until an error state is found. On the other hand, it should also be efficient to compute such that the overall performance of the model checking process is increased.

The area of directed model checking has recently found much attention, and various distance estimation functions have been proposed in this context [4, 6, 10, 13, 14, 18]. The basic principle to construct such functions is to first *abstract* the system under consideration, and then to use the length of an abstract error trace in this abstraction as an estimation for the actual length in the concrete. There are different strategies to define such distance functions. One way is to define abstractions that are coarse enough to find shortest abstract error traces in polynomial time (see, e. g., [4]). A different strategy is to choose an abstraction that is more fine-grained and does not admit polynomial algorithms for computing shortest abstract error traces. The distance estimate is then computed by *approximating* such error traces (see, e. g., [13]). Both strategies have proved to be successful for directed model checking.

In this paper, we introduce the *causal graph* structure to the context of directed model checking. For a given system Ξ, the causal graph is a dependency graph on the

component processes of Ξ that reflects how state changes in certain processes depend on state changes in others. Based on causal graph analysis, we first propose an adaptation of a distance function that has originally been introduced in the area of AI planning [7, 8]. We will see that this distance function follows the second strategy as outlined above. Furthermore, we propose a simple abstraction based on causal graph properties called *safe abstraction*, which is guaranteed not to introduce spurious error states (i. e., error traces found in this abstraction are guaranteed to correspond to error traces in the concrete system). We demonstrate that error detection is often significantly easier in this abstraction compared to the original system.

The structure of this paper is as follows. In Section 2, we give the basic notations and background needed for this work. Our contributions based on the causal graph are given in Sections 3 and 4, followed by an empirical experimental evaluation in Section 5. We conclude the paper and give an outlook on future work in Section 6.

2 Preliminaries

In this section, we define the notation and semantics for the systems considered in this paper, followed by an introduction to directed model checking.

2.1 Processes and systems

We model systems as parallel processes running in lockstep using global synchronization labels. Throughout the paper, let Σ be a finite set of synchronization labels (symbols). To distinguish between local states of an atomic process and the global state of the overall system, we use the term *location* for the former and *state* for the latter.

Definition 1 (process). *A process p is a labeled directed graph (L, T), where $L \neq \emptyset$ is the finite set of locations of p and $T \subseteq L \times \Sigma \times L$ is the set of local transitions of p.*

Whenever a given process performs a local transition from location l to l' with associated label $a \in \Sigma$, then all other processes must simultaneously perform a local transition with the same label a, or else the transition is not permitted. This gives rise to the following definition of the parallel composition of two processes. Parallel composition is an associative and commutative operation, up to isomorphism. For example, we can obtain $p_2 \parallel p_1$ from $p_1 \parallel p_2$ by renaming locations (l_1, l_2) to (l_2, l_1).

Definition 2 (parallel composition). *Let $p_1 = (L_1, T_1)$ and $p_2 = (L_2, T_2)$ be processes. The parallel composition of $p_1 \parallel p_2$ of p_1 and p_2 is the process (L, T) with $L = L_1 \times L_2$ and $T = \{((l_1, l_2), a, (l'_1, l'_2)) \mid (l_1, a, l'_1) \in T_1 \wedge (l_2, a, l'_2) \in T_2\}$.*

A *system* is simply the parallel composition of one or more processes. We choose this particular system model for ease of presentation; our basic ideas equally apply to other process models, such as ones involving internal transitions of processes or binary (rather than global) synchronization. Alternatively, such synchronization behaviour can also be modelled directly with our semantics. For example, to model asynchronous internal transitions of a process p, we can use a dedicated synchronization label $a_p \in \Sigma$

such that all internal transitions of p are labeled with a_p and all locations l of all other processes have transitions looping from l to l labeled with a_p. (More generally, such sets of loops can be used to model synchronization labels irrelevant to certain processes.)

Definition 3 (system). *A system is a pair $\Xi = ((p_1, \ldots, p_n), s_0)$, where $p_1, \ldots, p_n$ ($n \geq 1$) are processes called the components of Ξ. The parallel composition $P(\Xi) = p_1 \parallel \ldots \parallel p_n$ of the components is called the composite process of Ξ. Locations of $P(\Xi)$ are called states; we denote the states and transitions of $P(\Xi)$ by $S(\Xi)$ and $T(\Xi)$, respectively. The state $s_0 \in S(\Xi)$ is called the initial state of the system.*

A trace $\pi = s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n$ of Ξ is an alternating sequence of states and synchronization labels starting from the initial state such that $(s_{i-1}, a_{i-1}, s_i) \in T(\Xi)$ for all $i \in \{1, \ldots, n\}$. The length of a trace, $|\pi|$, is its number of transitions, i. e., $|\pi| = n$ for the given trace.

The problem we address in this paper, as in most work on directed model checking, is the *detection of error states* of a system, i. e., states reachable from the initial state which have an undesirable property. In CTL terms, this corresponds to proving the formula $\mathrm{E}\,\mathrm{F}\,\varphi$ where φ is a non-temporal formula that describes undesirable states. This is equivalent to the *falsification of invariants* of a system, i. e., to disproving the CTL formula $\mathrm{A}\,\mathrm{G}\,\neg\varphi$. In this paper, we consider the common situation where $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ is a conjunction of formulae where each formula φ_i describes properties of an individual component process p_i of the system. In this case, we can represent each conjunct φ_i by the set of locations of p_i that satisfy it.

Definition 4 (model checking task). *A model checking task is a pair $\Theta = (\Xi, L^*)$, where $\Xi = (((L_1, T_1), \ldots, (L_n, T_n)), s_0)$ is a system and $L^* = (L_1^*, \ldots, L_n^*)$ with $L_i^* \subseteq L_i$ for all $i \in \{1, \ldots, n\}$ denotes the target locations for each process of Ξ.*

An error trace of Θ is a trace of Ξ that ends in a state $s \in L_1^* \times \cdots \times L_n^*$.

To conclude this background section, we briefly remark that there is a close correspondence between finding error traces in our process model on the one hand and the nonemptiness problem for intersections of regular automata on the other hand. In this view, processes correspond to regular automata, the L_i^* sets correspond to accepting states, and parallel composition corresponds to language intersection. While this view is not necessarily useful for efficiently determining error traces in practical systems, it does show that deciding existence of error traces in a system is PSPACE-complete [11].

2.2 Directed Model Checking

Directed model checking is the approach of finding error states through an explicit state-space traversal guided by a distance estimation function $d^\#$. This function is computed fully automatically based on the declarative description of the system. In a nutshell, $d^\#$ is a function that maps states to natural numbers, reflecting an estimate of the shortest error distance. Typically, this estimate is the length of a corresponding abstract error trace. Each state encountered during a forward state-space traversal starting from the initial state is evaluated with $d^\#$, and states with lower values are preferred. Note that abstract distance functions only influence the *order* in which the states are explored,

```
1  function verify(Ξ, L*, d#):
2      s_0 = initial state of Ξ
3      open = empty priority queue
4      closed = ∅
5      priority = d#(s_0)
6      open.insert(s_0, priority)
7      while open is not empty:
8          s = open.pop-minimum()
9          if s satisfies φ(L*):
10             return False
11         closed = closed ∪ {s}
12         for each transition (s, a, s') ∈ T(Ξ):
13             if s' ∉ closed and s' ∉ open:
14                 priority = d#(s')
15                 open.insert(s', priority)
16     return True
```

Fig. 1. A basic directed model checking algorithm.

and hence completeness is *not* affected. On the one hand, it is desirable to have distance functions that are as informative as possible, so that only few states need to be explored until an error state is found. On the other hand, the computation of the distance estimate must not be too expensive.

Figure 1 shows a basic directed model checking algorithm. Given a model checking task (Ξ, L^*) and distance function $d^\#$, the algorithm returns *False* if there is a state that satisfies the error condition represented by L^*; otherwise it returns *True*. The initial state of Ξ is s_0. The algorithm maintains a priority queue *open* which contains visited, but not yet explored states. Through the method *open.pop-minimum*, the algorithm determines one such state s with minimum priority value (i. e., minimum estimated error distance) and removes it from the priority queue. This state s is then *expanded*, which is a three-step process. First, check if it is an error state; if so, we are done. Second, mark the state as explored by adding it to the *closed* set, so it will not be considered again later. Finally, determine the successor states of s and add them to the priority queue unless they have been encountered before. After expanding s, the process iterates with the new minimal element of *open*, until an error state is encountered or there remain no further states to check, at which point we can conclude that no error state can be reached.

This algorithm is known as *greedy search* (there are other algorithms like A^* for *optimal* search [17]; these are not considered in this paper). In a practical implementation of the algorithm, every state additionally stores information about how it has been reached, i. e., its immediate predecessor state and synchronization label at the time it was added to *open*. Therefore, if an error state s is finally reached, an error trace can be generated by back-tracing from s. Clearly, the efficiency of greedy search crucially depends on the quality of the estimates provided by $d^\#$. If these are perfect, the number of expansion steps of the algorithm is $n + 1$ where n is the length of the shortest error trace. On the other hand, if the estimates are completely uninformative, the algorithm degenerates to an unguided search algorithm such as depth-first search.

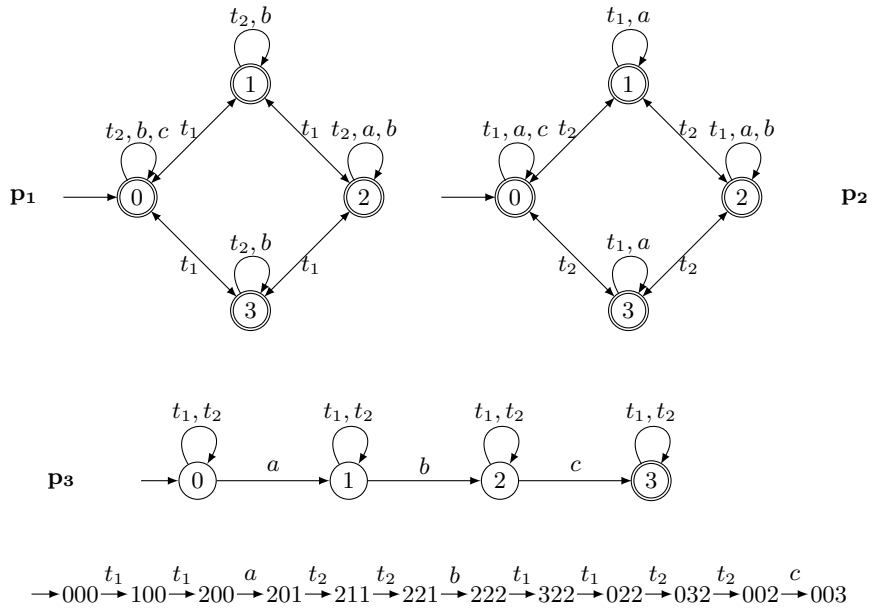

Fig. 2. An example system with three processes and a corresponding error trace. A transition with more than one label is an abbreviation for several parallel transitions, one for each label.

3 The Causal Graph

In this section, we introduce the central concepts of the causal graph heuristic, namely the *causal graph* and the *local subsystems* it induces. To provide some intuition for our definitions, we illustrate them with a running example (Fig. 2). The example system consists of three processes p_1, p_2 and p_3, each with locations $\{0, 1, 2, 3\}$ and transitions as shown in the figure. We assume that all processes are initially in location 0 and that we consider a state to be an error state iff process p_3 is in location 3 (i. e., $L_1^* = L_2^* = \{0, 1, 2, 3\}$ and $L_3^* = \{3\}$). The shortest error traces for this example have length 11 (one such error trace is also shown in Fig. 2), and indeed this is the distance estimate that the causal graph heuristic will assign in this case. However, other distance estimators considered in the directed model checking literature underestimate the true error distance:

- The d^L and d^U estimators [5, 6] measure the graph-theoretic distance to the nearest location L_i^* in each automaton p_i without taking into account synchronization labels. The d^L estimator maximizes over the individual distances, whereas d^U sums these values. In this case, we obtain $d^L(s_0) = \max\{0, 0, 3\} = 3$ and $d^U(s_0) = 0 + 0 + 3 = 3$ because only p_3 needs to move to a different location (3), which can be reached from location 0 in three steps.
- The h^L and h^U estimators [13] compute abstract error traces under the *monotonicity abstraction*. In the context of our running example, h^U considers an abstracted

problem where each process can "jump back to" a previously visited location at every step free of cost. In this case, we obtain $h^U(s_0) = 7$ because h^U fails to take into account that processes p_1 and p_2 must return to location 0 from location 2 in order to support the transition of p_3 from 2 to 3 via synchronization label c. The h^L estimator has the same weakness as h^U but additionally assumes in its abstraction that the required transitions of p_1 and p_2 from 0 to 2 can be performed simultaneously, leading to an estimate of $h^L(s_0) = 5$.

A common weakness of all these estimators, which causes the imperfect distance estimates, is that they fail to take into account that reaching a certain location of p_3 has a *side effect* on p_1 and p_2. In particular, they assume that as soon as p_3 has reached location 2, the error location 3 can be reached immediately in a single transition. The transition of p_3 from 2 to 3 requires p_1 and p_2 to follow a transition with label c, and the initial locations of p_1 and p_2 have outgoing transitions with this label from their locations in s_0, which is good enough for h^U and h^L (d^L and d^U do not care about synchronization at all). The estimators do not recognize that p_1 and p_2 must initially *move away* from location 0 (to location 2) before p_3 reaches location 2 in order to synchronize on the labels a (for p_1) and b (for p_2).

The causal graph heuristic overcomes this limitation by finding error traces in simple cases like this example *directly*, without further abstraction, while distances in "larger" systems are computed by combining information from smaller subsystems. To make this more precise, we must introduce the notion of causal graph. To motivate the following definition, observe that the labels $\{t_1, t_2, a, b, c\}$ play very different roles for the three processes in the example system:

- Label t_1 is very important for process p_1 because all proper (non-looping) transitions between locations of p_1 must synchronize on this label. We say that a label $a \in \Sigma$ *affects* a process (L, T) if $(l, a, l') \in T$ for some $l \neq l'$. In the example, t_1 affects p_1, t_2 affects p_2 and a, b and c affect p_3.
- Labels a and c do not cause non-looping transitions in p_1, but they are still relevant for the process because the current location of p_1 influences whether or not the overall system can synchronize on these labels. For example, the system cannot synchronize on a unless p_1 is in location 2. We say that a label $a \in \Sigma$ *restricts* a process (L, T) if there exists a location $l \in L$ such that for all $l' \in L$, $(l, a, l') \notin T$. In the example, a restricts p_1 and p_3, b restricts p_2 and p_3, and c restricts all processes.
- Finally, labels t_2 and b are completely irrelevant for process p_1: no matter in which location the process is, it can synchronize on these labels, and they cannot cause a change in location. We say that a label $a \in \Sigma$ is *irrelevant* for a process (L, T) if it does not affect or restrict the process. In the example, t_1 is irrelevant for p_2 and p_3, t_2 is irrelevant for p_1 and p_3, a is irrelevant for p_2, and b is irrelevant for p_1.

Using these different roles for labels and processes, we define the *causal graph* of a system Ξ as follows.

Definition 5 (causal graph). *The causal graph $CG(\Xi)$ of a system Ξ is the directed graph whose vertices are the component processes $p_1, \ldots, p_n$ of Ξ and which contains*

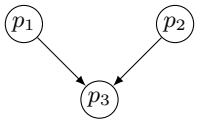

Fig. 3. The causal graph for the running example system.

an arc from p_i to p_j iff $i \neq j$ and there exists a label $a \in \Sigma$ that restricts or affects p_i and affects p_j.

The causal graph of the running example is shown in Fig. 3. Intuitively, the causal graph contains an arc from process p_i to p_j if there may be a need to change the location of p_i in order to change the location of p_j. To translate this intuition into a formal result, we first introduce the notion of *subsystems*.

Definition 6 (subsystem). *Let $\Xi = ((p_1, \ldots, p_n), s_0)$ be a system, let $\Theta = (\Xi, (L_1^*, \ldots, L_n^*))$ be a model checking task for Ξ, and let $P = \{p_{i_1}, \ldots, p_{i_k}\}$, $1 \leq i_1 < \cdots < i_k \leq n$ be a subset of the component processes of Ξ.*

The system $\Xi[P] := ((p_{i_1}, \ldots, p_{i_k}), (s_{0i_1}, \ldots, s_{0i_k}))$ is called the subsystem of Ξ induced by P, and the model checking task $\Theta[P] := (\Xi[P], (L_{i_1}^*, \ldots, L_{i_k}^*))$ is called the subtask of Θ induced by P.

It is easy to see that for any choice of P, $\Xi[P]$ is an *over-approximation* of Ξ: every trace π of Ξ induces a corresponding trace of $\Xi[P]$, which can be obtained from π by projecting all states to the components in P. Moreover, every error trace for Θ is an error trace for $\Theta[P]$. Of course, the converse is not true in general, and the existence of error traces for $\Theta[P]$ does not imply that there are error traces for Θ. However, there is a simple sufficient criterion under which all error traces of $\Theta[P]$ *do* correspond to error traces of Θ with the same synchronization sequence: namely, if P includes all processes with a non-trivial target location set (i. e., processes p_i for which not all locations are in the set L_i^*), as well as all causal graph *ancestors* of such processes. This is essentially the idea of *cone-of-influence reduction* [2].

In fact, cone-of-influence reduction is still error-preserving if we consider an alternative definition of causal graphs where we only introduce an arc from p_i to p_j if some label *restricts* p_i and affects p_j. The reason why we also include arcs from p_i to p_j if some common label *affects* both of them is that this gives an additional decomposition result, which we will discuss in Section 4.1.

As a side remark, under our definition, if the causal graph consists of more than one weakly connected component, then there exists an error trace iff each subtask induced by a weakly connected component has an error trace. (The overall error trace is then essentially the concatenation of these "subtraces".) The intuitive reason for this property is that if two sets of processes P and P' are causally disconnected, then all state transitions that affect the locations of processes in P are not restricted by or affect the locations of processes in P' (and vice versa), and hence the corresponding subtasks can be addressed independently. A similar decomposition result does not hold under the alternative definition of causal graphs, where traces that affect the processes P are not restricted by the processes P', but can still change the locations of P' as a side effect.

4 The Causal Graph Heuristic

The causal graph heuristic estimates the cost of reaching an error state by computing distance estimates for a number of subtasks which are derived by looking at small "windows" of the causal graph. In this section, we describe this procedure conceptually as a bottom-up computation along a topological sorting of the causal graph. (In a practical implementation, a top-down implementation is more efficient, but both approaches lead to the same distance estimates.) Since we require a topological sorting of the causal graph, the procedure only works for acyclic causal graphs; we will later explain how to deal with the cyclic case. For now, let us just remark that deciding the existence of error traces is already PSPACE-complete for systems with acyclic causal graphs, even under the further restriction that all processes have only two locations [1].

Throughout this section, we assume that we are given a model checking task $\Theta = (\Xi, (L_1^*, \ldots, L_n^*))$ for a system $\Xi = ((p_1, \ldots, p_n), s_0)$, and that our objective is to compute a distance estimate for a given state s of Ξ, which we denote as $h^{CG}(s)$. For each process $p_i = (L_i, T_i)$ and each pair of locations $l_i, l_i' \in L_i$, the causal graph heuristic computes a distance estimate $\mathit{cost}_{p_i}(l_i, l_i')$ for the cost of changing the location of p_i from l_i to l_i'. The overall distance estimate of s is then defined as the sum of the costs of reaching the nearest error location in each process, i. e., $h^{CG}(s) = \sum_{i=1}^{n} \min_{l_i^* \in L_i^*} \mathit{cost}_{p_i}(l_i, l_i^*)$ for $s = (l_1, \ldots, l_n)$. Note that the d^U estimate, due to Edelkamp et al. [5, 6], is defined by the same equation, but using a different estimate for $\mathit{cost}_{p_i}(l_i, l_i^*)$, which is simply the graph-theoretic distance from l_i to l_i^*. In contrast, the cost estimates for h^{CG} take synchronization labels into account and usually provide larger (and, as we shall see in the experimental evaluation in Section 5, more accurate) estimates than the graph-theoretic distance.

4.1 Independent Processes

In this section and the following, we describe how the $\mathit{cost}_p(l, l')$ estimates are computed. We begin with the case where process p has no predecessors in the causal graph. We call such a process *independent* because (by the definition of causal graphs) it can change location independently of and without affecting the locations of other processes.

Let $p = (L, T)$ be an independent process. In this case, like in the case of the d^U heuristic, we define $\mathit{cost}_p(l, l')$ as the graph-theoretic distance from l to l' in p. For independent processes, this is an appropriate definition because local transitions are not restricted by any other processes. Hence, in any state of the system, a sequence of synchronization labels leading from l to l' does correspond to an executable trace that changes the location of p from l to l', without affecting the locations of other processes.

In our running example, Fig. 3 shows that processes p_1 and p_2 do not have predecessors in the causal graph, and indeed, Fig. 2 shows that these are independent processes, as the only labels affecting them – t_1 for p_1, t_2 for p_2 – are irrelevant for the other processes. Therefore, the cost estimates for these processes equal the graph distances (e. g., $\mathit{cost}_{p_1}(0, 2) = 2$ and $\mathit{cost}_{p_2}(2, 3) = 1$).

Safe Abstraction For some independent processes, there is actually no need to compute any cost estimates at all. Consider the case where p is independent and all cost

estimates for p are finite (or, equivalently, p is strongly connected). Without loss of generality, we assume that

- the target location set for p is not empty (otherwise, trivially there exist no error states, since the error states are formed as the Cartesian product of the target location sets of the processes), and
- each label a that occurs in a transition of any process also occurs in a transition of p (otherwise transitions with label a can never be synchronized, and we can remove all such transitions in a preprocessing step).

In this case, it is possible to separate the local transitions for p from the rest of the model checking task completely. Let $\Xi[P']$ be the subsystem induced by all processes of Ξ except p, i. e., $P' = \{p_1, \ldots, p_n\} \setminus \{p\}$. Given a state s of Ξ and an error trace π' for $\Xi[P']$ that starts in the projection of s to P', we can compute an error trace for Ξ starting from s with a simple polynomial algorithm:

- If $|\pi'| = 0$ (i. e., π' is the empty trace), then s' is already an error state for $\Xi[P']$, and hence all processes except possibly p are in a target location in state s. Because p is strongly connected and has at least one target location, we can find a sequence of local transitions of p that lead from its location in s to a target location of p. Because p is independent, these transitions are not restricted by the other processes and do not affect their locations. By following these local transitions, we can go from state s to a global error state.
- If $|\pi'| = n \geq 1$, then the trace starts with some global transition (s', a, t') of $\Xi[P']$. Because p is strongly connected and has at least one location with an outgoing transition labeled a, we can find a sequence of local transitions of p that lead from its location in s to a location in which p can synchronize on a. Because p is independent, these transitions are not restricted by the other processes and do not affect their locations. By following these local transitions, we can go from state s to a state $\tilde{s}$ whose projection to P' is s' and in which all processes can synchronize on a, and from there to a state t whose projection to P' is t'. Since t' starts an error trace of length $n - 1$ in $\Xi[P']$, we can reach an error state of Ξ from t (and hence, from s) by an inductive argument.

The analysis shows that if the independent process p is strongly connected, there exists a *safe abstraction* of Ξ to P': any error trace of $\Xi[P']$ induces an error trace of Ξ, and of course the converse is also true because subsystems are always over-approximations.

Under these circumstances, we can run the directed model checking algorithm directly on $\Xi[P']$ instead of Ξ, and then apply the above procedure to convert the error trace for the abstracted problem into a concrete one. Of course, the abstraction may cascade, as $\Xi[P']$ may admit further safe abstractions, even for processes that were not originally independent in P. In our experimental analysis (Section 5), we will present results for the causal graph heuristic both with and without safe abstraction.

We briefly remark that in our running example, we could safely abstract away p_1 and p_2 since these processes are independent and strongly connected. However, as we will now turn to the question of computing cost estimates for non-independent processes,

```
1  function compute-costs(Ξ, s, p, l):
2      Let pred(p) be the set of immediate predecessors of p in CG(Ξ).
3      (L, T) := p
4      cost_p(l, l) := 0
5      cost_p(l, l') := ∞ for all l' ∈ L \ {l}
6      context(l, p_i) := location of p_i in s, for all p_i ∈ pred(p)
7      unreached := L
8      while unreached contains a location l' ∈ L with cost_p(l, l') < ∞:
9          Choose such a location l' ∈ unreached minimizing cost_p(l, l').
10         unreached := unreached \ {l'}
11         for each transition (l', a, l'') ∈ T from l' to some l'' ∈ unreached:
12             target-cost := cost_p(l, l') + 1
13             target-context := ∅
14             for each process p_i = (L_i, T_i) ∈ pred(p):
15                 m := context(l', p_i)
16                 Choose (m', m'') ∈ L_i × L_i such that (m', a, m'') ∈ T_i
                       and cost_{p_i}(m, m') is minimized.
17                 target-cost := target-cost + cost_{p_i}(m, m')
18                 target-context(p_i) := m''
19             if target-cost < cost_p(l, l''):
20                 cost_p(l, l'') := target-cost
21                 context(l'', p_i) := target-context(p_i) for all p_i ∈ pred(p)
22     return cost_p(l, l') for all l' ∈ L
```

Fig. 4. Modified Dijkstra algorithm for computing $cost_p(l, l')$.

we will assume for the rest of this section that safe abstraction is not performed on the running example.

4.2 Processes with Causal Predecessors

For processes p which do have predecessors in the causal graph, cost estimates are also computed by searching for paths in the labeled directed graph defined by the process. However, here we improve on the d^U approach by taking into account the synchronization labels on the local transitions: in addition to counting the number of local transitions of p required to reach a given location, we also consider the costs for moving the other processes of the system into locations which can synchronize with these transitions. Note that by the definition of causal graphs, the *only* processes which can potentially restrict the non-looping local transitions of p are its causal predecessors, which we denote as $pred(p)$. Because we compute costs in a bottom-up order along a topological sorting of the causal graph, we have already computed all cost estimates for these processes. Hence, the computation of $cost_p(l, l')$ is based on finding traces from l to l' in the subsystem of Ξ induced by $\{p\} \cup pred(p)$, taking into account the known cost estimates for the processes $pred(p)$.

The algorithm for computing the cost values $cost_p(l, l')$ is shown in Fig. 4. It is a modification of Dijkstra's algorithm for finding shortest paths in weighted directed graphs, applied to the process $p = (L, T)$. Like Dijkstra's algorithm, it is a one-to-all

procedure, i. e., for a given start location l, it computes $\mathit{cost}_p(l, l')$ for all $l' \in L$. The only difference to Dijkstra's algorithm is that we do not define the cost of a transition $(l', a, l'') \in T$ before applying the algorithm. Instead, the transition cost is computed as soon as location l' is expanded by the algorithm, and it depends on the current locations of $\mathit{pred}(p)$ in the situation where l' is reached.

In detail, the cost of reaching $l'' \in L$ through transition $(l', a, l'') \in T$ is computed as the cost of reaching l' plus the *setup cost* required to take $\mathit{pred}(p)$ into locations that allow synchronization on the label a, plus 1 for taking the actual transition with label a that takes p from l' to l'' (lines 12–18). To estimate the setup cost for each predecessor $p_i \in \mathit{pred}(p)$, we associate each location $l' \in L$ with locations $\mathit{context}(l', p_i)$ for each $p_i \in \mathit{pred}(p)$, with the interpretation that when $l' \in L$ is first reached, we assume that process p_i is in location $m = \mathit{context}(l', p_i)$. The setup cost for a given process is then the cheapest cost, according to the previously computed cost_{p_i} values, for taking process p_i from m to a location m' where it can synchronize on the label a (lines 15–17).

If it turns out that $(l', a, l'') \in T$ reaches l'' more cheaply than the previously considered transitions (line 19), then the cost of l'' is updated accordingly (line 20, as in Dijkstra's algorithm). At the same time, the context of l'' is set so that it reflects the way in which we have reached the location: by performing appropriate setup transitions for $\mathit{pred}(p)$ and then synchronizing on label a (lines 16, 18, 21).

We remark that the algorithm is not guaranteed to find a globally shortest trace in the subsystem induced by $\{p\} \cup \mathit{pred}(p)$. Indeed, it may fail to find *any* path to a given location $l' \in L$ even though it is reachable. The reason for this is that the setup for each transition (l', a, l'') is performed greedily, without backtracking on the choice of *how* to modify the current context in order to allow synchronization on label a: we always pick a *locally cheapest* setup sequence. While it would of course be preferable to guarantee the success of the *compute-costs* algorithm, unfortunately this is not possible to do in polynomial time if $P \neq NP$: if we could, this would decide the existence of error traces in the model checking task induced by $\{p\} \cup \mathit{pred}(p)$. However, it is known that error detection for the subtask induced by a single process and its direct causal graph predecessors is NP-complete [7].

Returning to our running example, the algorithm computes the following cost estimates $\mathit{cost}_{p_3}(0, l')$ for the state $(0, 0, 0)$:

- $\mathit{cost}(p_3)(0, 0) = 0$: This is due to the initialization step (line 4).
- $\mathit{cost}(p_3)(0, 1) = 0 + 1 + 2 = 3$: the three terms correspond to the cost of location 0, the constant term 1, and the setup cost to reach locations of p_1 and p_2 in which we can synchronize on label a. In this case, we need to change p_1 from location 0 to 2, for a setup cost of 2.
- $\mathit{cost}(p_3)(0, 2) = 3 + 1 + 2 = 6$: cost of location 1, constant term 1, setup cost to reach locations of p_1 and p_2 in which we can synchronize on label b. In this case, we need to change p_2 from location 0 to 2, for a setup cost of 2.
- $\mathit{cost}(p_3)(0, 3) = 6 + 1 + 4 = 11$: cost of location 2, constant term 1, setup cost to reach locations of p_1 and p_2 in which we can synchronize on label c. In this case, we need to change both processes from location 2 to 0, for a setup cost of $2 + 2$.

4.3 Causal Graphs with Cycles

Up to this point, we have given a complete description of how to compute $h^{CG}(s)$ for systems with acyclic causal graphs. Unfortunately, many practical systems tend to have causal graphs with cycles. In this work, we use a rather simple idea to extend the definition of the heuristic to the general case (for an alternative approach, see Section 6).

If $CG(\Xi)$ is not acyclic, we impose a total order $p'_1 \prec \cdots \prec p'_n$ on the processes of Ξ. The computation of cost values then proceeds as previously described, except that for process p'_i, the *compute-costs* function does not consider *all* causal predecessors $pred(p'_i)$ of p'_i, but only those which are ordered before p'_i in the ordering. Semantically, this means that we do not consider the synchronization costs for *all* processes, but only a subset of them. Of course, different total orders lead to different synchronization aspects being respected by this abstraction, so in practice one would prefer an order which is "close" to a topological sorting in some sense (e. g., loses as few arcs of the causal graph as possible). In our experiments, we use some simple greedy criteria to compute a reasonable ordering (see Section 5.1).

5 Evaluation

We implemented the causal graph heuristic and the safe abstraction technique from Section 4 in the model checker MCTA [15] and evaluated it on a number of academic and industrial benchmarks. The experimental results were obtained on a system with a 3 GHz Intel Pentium 4 CPU, using a memory bound of 1 GB. We compare h^{CG} with the other distance functions d^L, d^U [5, 6], h^L and h^U [13] as implemented in MCTA.

5.1 Implementation Details

Our benchmark models consist of parallel automata with interleaving and binary synchronization semantics. This easily fits into the process model used throughout this paper. In addition, some benchmarks feature bounded integer variables and (unbounded) clock variables. Edges in the automata can be guarded by integer or clock constraints, and edges can also reset clock variables and set integers to new values as effects.

The h^{CG} heuristic as implemented in MCTA directly reflects integer and location variables, whereas clocks are ignored for the distance computation. (In fact, abstracting clocks away is the easiest way to deal with them for the computation of distance functions and has already successfully been done in other approaches [4, 13].) Essentially, each automaton and each bounded integer variable is identified with a process p in the sense of Definition 1. Both kinds of processes can be subject to safe abstraction as described in Section 4.1; however, as clocks are ignored by the distance computation, to ensure safety we additionally check that these processes do not affect clock variables.

For systems with cyclic causal graphs, we greedily impose an ordering on the processes such that as much as possible of the important synchronization behaviour is respected. Essentially, arcs in the causal graph are preferably ignored if they are induced by as few system transitions as possible. Furthermore, as processes that correspond to automata play a dedicated role in the system, we order them after processes that correspond to integer variables. In more detail, we require for all processes p, p' that if p corresponds to an automaton and $p \prec p'$, then p' also corresponds to an automaton.

Table 1. Experimental results in terms of number of explored states and search time for the heuristics d^L, d^U, h^L, h^U in comparison to h^{CG} and h^{CG} with safe abstraction (denoted with h^{CG}_{safe}). Dashes indicate exhaustion of memory ($>$ 1 GB).

	explored states						search time in seconds					
Inst.	d^L	d^U	h^L	h^U	h^{CG}	h^{CG}_{safe}	d^L	d^U	h^L	h^U	h^{CG}	h^{CG}_{safe}
C_1	18796	16817	1928	715	5129	5129	0.1	0.1	0.1	0.0	0.5	0.5
C_2	66389	61229	4566	1612	6268	2721	0.4	0.4	0.1	0.1	0.6	0.4
C_3	94536	85332	6002	734	6943	3241	0.6	0.6	0.2	0.1	0.6	0.4
C_4	1.11e+6	1.04e+6	81131	9120	57493	6201	6.8	6.3	1.7	0.3	1.1	0.3
C_5	1.27e+7	1.21e+7	430494	83911	494778	13675	76.3	74.7	9.2	2.1	9.3	0.5
C_6	–	–	4.56e+6	718015	5.54e+6	24125	–	–	83.1	12.4	68.4	0.9
C_7	–	–	–	2.55e+6	–	57595	–	–	–	41.4	–	2.3
C_8	–	–	–	–	–	122880	–	–	–	–	–	6.5
C_9	–	–	–	–	–	379981	–	–	–	–	–	24.2
M_1	12277	185416	4581	7668	6245	6245	0.3	6.1	0.1	0.1	0.1	0.1
M_2	43784	56240	15832	18847	18988	8472	0.6	0.8	0.2	0.2	0.2	0.2
M_3	54742	869159	7655	19597	27365	10632	0.8	398.0	0.1	0.2	0.4	0.2
M_4	202924	726691	71033	46170	96418	18574	3.4	110.5	0.8	0.5	1.4	0.4
N_1	15732	10215	50869	9117	8171	8171	0.4	0.2	2.7	0.1	0.2	0.2
N_2	102909	642660	30476	23462	30540	30540	3.0	239.6	0.6	0.5	0.8	0.8
N_3	131202	1.16e+6	11576	43767	40786	40786	4.1	2342.2	0.2	0.9	1.1	1.1
N_4	551091	330753	100336	152163	252558	252558	24.0	11.7	2.1	3.7	9.5	9.5
F^A_5	271	271	9	9	11	11	0.0	0.0	0.0	0.0	0.0	0.0
F^A_{10}	271	271	9	9	11	11	0.0	0.0	0.0	0.0	0.0	0.0
F^A_{15}	271	271	9	9	11	11	0.0	0.0	0.0	0.0	0.0	0.0
F^B_5	496	9	179	7	9	9	0.0	0.0	0.0	0.0	0.0	0.0
F^B_{10}	–	9	86378	7	9	9	–	0.0	2.1	0.0	0.0	0.0
F^B_{15}	–	9	–	7	9	9	–	0.0	–	0.0	0.0	0.0
A_2	27	23	36	25	13	13	0.0	0.0	0.0	0.0	0.0	0.0
A_3	344	296	206	82	199	199	0.0	0.0	0.0	0.0	0.0	0.0
A_4	38209	19034	76811	39	179	179	0.5	0.3	12.9	0.1	0.1	0.1
A_5	–	–	263346	4027	188499	188499	–	–	90.8	3.1	90.8	90.7
A_6	–	–	–	–	–	–	–	–	–	–	–	–

5.2 Benchmarks

Our benchmarks stem from the AVACS[1] benchmark suite. The M and N examples ("Mutual Exclusion") are industrial benchmarks which come from a case study that models a real-time protocol to ensure mutual exclusion of a state in a distributed system via asynchronous communication. The protocol is described in full detail by Dierks [3]. The C examples ("Single-tracked Line Segment") stem from a case study from an industrial project partner of the UniForM project [12] where the problem is to design a distributed real-time controller for a segment of tracks where trams share a piece of track. For the evaluation of our approach we chose the property that both directions are never given simultaneous permission to enter the shared segment. In both case stud-

[1] `http://www.avacs.org/`

Table 2. Experimental results in terms of error trace length for the heuristics d^L, d^U, h^L, h^U in comparison to h^{CG} and h^{CG} with safe abstraction (denoted with h^{CG}_{safe}). Dashes indicate exhaustion of memory ($>$ 1 GB). Abbreviations: #a: number of parallel automata, #vars: number of integer and clock variables, #safe: number of variables removed by safe abstraction. For h^{CG}_{safe}, trace lengths reported as $x + y$ denote trace length x for the abstract error trace and $x + y$ for the concrete error trace.

				error trace length					
Instance	#a	#vars	#safe	d^L	d^U	h^L	h^U	h^{CG}	h^{CG}_{safe}
C_1	5	15	0	1167	1058	100	73	118	118
C_2	6	17	1	1847	1674	132	99	169	118 + 0
C_3	6	18	1	2153	1214	128	86	167	118 + 0
C_4	7	20	2	6805	2949	344	139	354	125 + 5
C_5	8	22	3	35067	11696	1057	300	1034	125 + 9
C_6	9	24	4	–	–	3217	864	4167	132 + 14
C_7	10	26	5	–	–	–	2412	–	139 + 19
C_8	10	27	5	–	–	–	–	–	132 + 16
C_9	10	28	5	–	–	–	–	–	192 + 30
M_1	3	15	0	2779	106224	457	71	231	231
M_2	4	17	1	11739	13952	1124	119	395	240 + 3
M_3	4	17	1	12701	337857	748	124	361	205 + 4
M_4	5	19	2	51402	290937	3381	160	642	219 + 7
N_1	3	18	0	3565	2669	26053	99	243	243
N_2	4	20	0	18180	415585	1679	154	376	376
N_3	4	20	0	20021	262642	799	147	232	232
N_4	5	22	0	90467	51642	2455	314	478	478
F^A_5	6	6	0	218	218	8	8	8	8
F^A_{10}	11	11	0	218	218	8	8	8	8
F^A_{15}	16	16	0	218	218	8	8	8	8
F^B_5	5	6	0	79	6	12	6	6	6
F^B_{10}	10	11	0	–	6	22	6	6	6
F^B_{15}	15	16	0	–	6	–	6	6	6
A_2	8	0	0	22	13	21	21	12	12
A_3	16	0	0	169	39	24	18	24	24
A_4	32	0	0	867	129	42	28	36	36
A_5	64	0	0	–	–	112	47	56	56
A_6	128	0	0	–	–	–	–	–	–

ies, a subtle error has been inserted by manipulating a delay so that the asynchronous communication between these automata is faulty.

The F^A and F^B examples are flawed versions of the *Fischer protocol* for mutual exclusion (cf. [16]). The difference between F^A and F^B is in the way they encode the error condition.

As a final set of benchmarks, we use the *arbiter trees* case study, which models a mutual exclusion protocol based on a tree of binary arbiters [19]. Client processes are situated at the leaves of the tree. The benchmarks A_2–A_6 contain arbiter trees of height 2–6, with an exponentially growing number of processes.

5.3 Results

We compare the h^{CG} heuristic and the safe abstraction technique based on causal graph analysis with the heuristics d^L, d^U [5, 6], h^L and h^U [13] as implemented in MCTA. We compare the number of explored states, the search time in seconds (Table 1) and the length of the found error traces (Table 2). Table 2 also gives additional information about the benchmark models, such as the number of parallel processes and the number of processes removed by safe abstraction.

The results show that our distance function is competitive with the previous approaches. In addition, safe abstraction leads to significantly better performance when applicable. We observe that the h^{CG} heuristic is much more accurate than the d^L and d^U heuristics. Due to better guidance, significantly fewer states are explored until an error state is found, leading to much better overall performance in most cases. Moreover, the error traces found by h^{CG} are significantly shorter than those obtained by d^L and d^U. This significant improvement is particularly interesting because of the connection between h^{CG} and d^U (recall that for independent processes, the cost estimates of h^{CG} and d^U are equal). The experimental results further show that h^{CG} is competitive with h^L, although somewhat less informed than h^U.

Considering the results for safe abstraction, we observe that in models that contain independent variables, the model reduction obtained by safe abstraction leads to a significant performance gain with h^{CG}. Moreover, the computational overhead to find such variables is low (a fraction of a second).

6 Conclusions

We have introduced the causal graph structure to directed model checking and demonstrated it to be a useful concept for error detection. We have adapted a distance estimation function from AI planning based on causal graph analysis, which is competitive with other distance heuristics in MCTA. Further, we presented an abstraction with the property that reachable abstract error states are guaranteed to correspond to reachable error states in the original system. We have shown that such safe abstractions can significantly improve the overall performance of a directed model checking algorithm when applicable, while requiring very little preprocessing overhead when not applicable.

In the future, it will be interesting to consider further extensions of the causal graph concept, in particular the question of how to deal with cycles in the causal graph more directly (see also [9]). In contrast to the approach presented in this paper, where cycles are resolved through a statically imposed ordering of processes, this could also be done *dynamically* during search. Furthermore, there seems to be potential to consider "larger" local subproblems than we have done, in order to improve the precision of the h^{CG} estimator. We expect that these approaches will allow further advances in the practical performance of directed model checking approaches.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `http://www.avacs.org/`).

References

1. Ronen I. Brafman and Carmel Domshlak. Structure and complexity in planning with unary operators. *Journal of Artificial Intelligence Research*, 18:315–349, 2003.
2. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.
3. Henning Dierks. Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.
4. Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *LNCS*, pages 19–34. Springer-Verlag, 2006.
5. Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2):247–267, 2004.
6. Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *LNCS*, pages 57–79. Springer-Verlag, 2001.
7. Malte Helmert. A planning heuristic based on causal graph analysis. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 161–170. AAAI Press, 2004.
8. Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.
9. Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*. AAAI Press, 2008.
10. Jörg Hoffmann, Jan-Georg Smaus, Andrey Rybalchenko, Sebastian Kupferschmid, and Andreas Podelski. Using predicate abstraction to generate heuristic functions in Uppaal. In Stefan Edelkamp and Alessio Lomuscio, editors, *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, volume 4428 of *LNCS*, pages 51–66. Springer-Verlag, 2007.
11. Dexter Kozen. Lower bounds for natural proof systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 254–266. IEEE Computer Society, 1977.
12. Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, and Alexander Baer. The UniForM workbench, a universal development environment for formal methods. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*, volume 1709 of *LNCS*, pages 1186–1205. Springer-Verlag, 1999.
13. Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *LNCS*, pages 35–52. Springer-Verlag, 2006.
14. Sebastian Kupferschmid, Jörg Hoffmann, and Kim G. Larsen. Fast directed model checking via russian doll abstraction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 203–217. Springer-Verlag, 2008.
15. Sebastian Kupferschmid, Martin Wehrle, Bernhard Nebel, and Andreas Podelski. Faster than UPPAAL? In Aarti Gupta and Sharad Malik, editors, *Proceedings of the 20th International*

Conference on Computer Aided Verification (CAV 2008), volume 5123 of *LNCS*, pages 552–555. Springer-Verlag, 2008.
16. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.
17. Judea Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley, 1984.
18. Kairong Qian and Albert Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 497–511. Springer-Verlag, 2004.
19. Charles L. Seitz. Ideas about arbiters. *Lambda*, 1:10–14, 1980.