University of Basel

Under-Approximation Refinement for Timed Automata

Bachelor's thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
http://ai.cs.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Martin Wehrle

Kevin Grimm
kevin.grimm@unibas.ch
12-058-053

12.01.2017

Acknowledgements

I would like to thank my supervisor Dr. Martin Wehrle for all the help and advice he gave me during the weekly meetings. He also provided me with a lot of useful background information about Mcta which made an implementation of the algorithm possible.
Also I would like to thank Prof. Dr. Malte Helmert for providing me with the opportunity to work on this Bachelor's thesis.

Abstract

Validating real-time systems is an important and complex task which becomes exponentially harder with increasing sizes of systems. Therefore finding an automated approach to check real-time systems for possible errors is crucial. The behaviour of such real-time systems can be modelled with timed automata.
This thesis adapts and implements the under-approximation refinement algorithm developed for search based planners proposed by Heusner et al. to find error states in timed automata via the directed model checking approach. The evaluation compares the algorithm to already existing search methods and shows that a basic under-approximation refinement algorithm yields a competitive search method for directed model checking which is both fast and memory efficient. Additionally we illustrate that with the introduction of some minor alterations the proposed under-approximation refinement algorithm can be further improved.

Table of Contents

1 Introduction

Real-time systems are often used for safety related tasks. It is crucial that such systems are validated to contain no errors. For example, the embedded system that controls an air-bag has to trigger the activation of the air-bag during a small time frame. It is not possible for the system to fulfil its potentially life-saving purpose if the explosion of the air-bag is triggered to early or even too late. Searching for bugs can be done manually for small systems but is often a challenging task for systems of practical size. To automate the validation the real-time systems have to be modelled.
Timed automata are an extension of finite automata and can be used to model real-time systems [1]. To automatically find error states in timed automata the directed model checking approach can be used [2–4]. With the help of a search algorithm and heuristics the directed model checking approach expands the state space of a system to find error states. A major problem of directed model checking is the state explosion problem [5]. It is caused by the fact that the state space grows exponentially in size compared to the task. There are many proposed solutions for this problem and many of them exploit the observation that the used operators to find an error state are often just a small subset of all available operators [6, 7]. One of these so called transition-based algorithms is the under-approximation refinement algorithm by Heusner et al. [8].
To keep the state space small the proposed algorithm restricts the available operators and iteratively adds more if needed. This approach leads to good results in classical planning and could therefore also be a viable option to the similar field of directed model checking.

In this thesis we first cover the background for adapting the under-approximation refinement algorithm for directed model checking of timed automata. In a second step we define an under-approximation and describe a basic algorithm which implements the under-approximation refinement idea for concurrent systems of timed automata. We then evaluate the algorithm with several benchmark sets and compare the results to other directed model checking algorithms. In a last part we propose a small alteration to the previously implemented under-approximation refinement algorithm and evaluate the results.

2 Preliminaries

This chapter introduces the background needed for this thesis. First, we describe timed automata and provide a fundamental definition. We also present a greedy best-first search algorithm which serves as the fundamental structure for the under-approximation refinement algorithm. In the last part of this chapter we show the concept of directed model checking and introduce the verification tool Mcta [9] in which the under-approximation algorithm is later implemented.

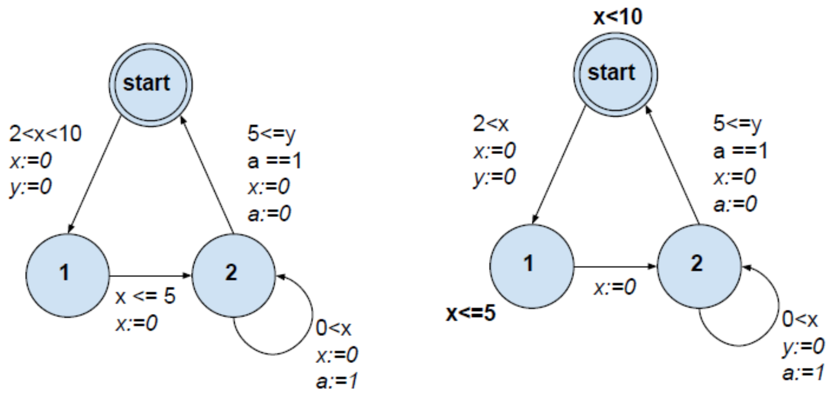
2.1 Timed Automata

Timed automata are an extension of finite automata [1]. They are used to model real-time systems. Like a finite automaton, a timed automaton consists of locations, edges and a set of inputs. Additionally, a timed automaton is extended with a finite set of real-valued variables called *clocks*. This addition allows the modelling of real-time based systems.

Clock values are initially set to zero when the system is started. All clocks increase simultaneously at the same speed while the system is running. With the clock variables also *clock constraints* are introduced. An edge can only be taken if the clock values satisfy the clock constraints of the corresponding edge. Additionally, we use a definition of timed automata extended by bounded *integer variables*. A timed automaton features a set of integer variables which have an integer value and a domain assigned. Each edge can have an integer guard consisting of *integer constraints*. An edge can only be taken if the integer constraints and the clock constraints are satisfied. If an edge is taken, the edge effects are triggered. These effects can reset a subset of clock values to zero and also assign new integer values to a subset of the integer variables [1, 10].

2.1.1 Example

In Fig. 2.1a, a timed automaton is initiated with two clock variables x and y. The automaton also has an integer variable a which is initially set to 0. The automaton starts in the location *start* and has two options to proceed. Either it makes a *delay transition* and therefore stays in the same location only increasing the clock values or it takes an edge if the constraints allow it and makes an *action transition*.
The first action transition can be taken as long as the clock variable x has a value between 2 and 10. While this constraint is not satisfied the automaton has to make delay transitions. If the first edge is taken, the values of x and y are set to 0. It is also possible that none or just a subset of the clock variables are reset as can be seen in the edge connecting the locations 1 and 2. The edge between the location 2 and *start* shows an integer constraint which requires that a has the value 1. This edge also has an integer-assignment effect that sets the integer variable a to 0.



a) Timed Automaton b) Timed Safety Automaton

Figure 2.1: Example of timed automata with two clock variables x and y and one integer variable a.

It is important to note that both clock constraints and integer constraints are only enabling and not forcing, thus the automaton could just be idling in any location while only taking delay transitions. This problem can be solved by introducing local-timing constraints called *location invariants*.
For example in Fig. 2.1b, location invariants are introduced. Therefore the timed automaton is not allowed to idle in any location while only taking delay transitions. This leads to the invariant conditions that the *start* location has to be left before x reaches the value 10 and the location 1 has to be left while the clock variable x is smaller or equal to the value of 5.
The definition of timed automata with location invariants is often referred to as timed safety automata [1]. For simplicity we use the term timed automata in this thesis synonymously for timed automata with location invariants.

2.1.2 Syntax

There are many definitions for timed automata. We use a definition and notation similar to the one of Bengtsson and Yi [1] extended by integer variables.
A *clock constraint* is a formula of the form $x \sim n$ or $x - y \sim n$, where $\{x, y\} \in C$ are clock variables, $n \in \mathbb{N}$ and $\sim \in \{<, \leq, ==, \geq, >\}$.
An *integer constraint* is a formula of the form $v_1 \sim n$ or $v_1 \sim v_2$, where $\{v_1, v_2\} \in V$ are integer variables, $n \in \mathbb{N}$ and $\sim \in \{<, \leq, ==, \geq, >\}$.
An *integer assignment* is an expression of the form $v_1 := n$, where $v_1 \in V$ is an integer variable, $\mathrm{dom}(v_1)$ is the domain of v_1 and $n \in \mathrm{dom}(v_1)$ is an integer value.
A *clock reset* is an expression of the form $x := 0$, where $x \in C$ is a clock variable.
A *location invariant* is a conjunction of clock constraints of the form $x \sim n$, where $x \in C$ is a clock variable, $n \in \mathbb{N}$ and $\sim \in \{<, \leq\}$.
For a set Y, the powerset of Y is denoted with 2^Y.

Definition 2.1. A timed automaton is a 7-tuple $\langle L, \Sigma, C, V, E, I, l_0 \rangle$, where

- L is a finite set of locations,
- Σ is a finite alphabet standing for synchronisation labels,
- C is a finite and real-valued set of variables standing for clocks,
- V is a set of bounded integer variables with a domain $\mathrm{dom}(v_i)$ for each $v_i \in V$,
- $E \subseteq L \times \Sigma \times B(C) \times B(V) \times 2^C \times IA(V) \times L$ is a set of edges, where
  - $B(C)$ is a set of clock constraints,
  - $B(V)$ is a set of integer constraints,
  - 2^C is a set of clock resets and
  - $IA(V)$ is a set of integer assignments,
- $I : L \rightarrow LV(C)$ assigns location invariants, where $LV(C)$ is a set of location invariants, and
- $l_0 \in L$ is the initial location.

An edge can be written as $l \xrightarrow{cg,ig,a,i,r} l'$ with a clock guard $cg \subseteq B(C)$, an integer guard $ig \subseteq B(V)$, an action $a \in \Sigma$, a set of integer assignments $i \subseteq IA(V)$ and a set of clock resets $r \subseteq 2^C$.
The synchronisation labels in Σ are used for running concurrent systems of timed automata as explained later.

2.1.3 State Space

To model the behaviour of timed automata we use *state spaces.* A state space of a timed automaton consists of states and two types of transitions. The first transition is called a delay transition, which has no other effects than increasing the clock values, and retaining locations and integer values. The other transition type is the action transition which can have the effect of integer assignments, clock resets and location changes.

The following definition of the exact state space is based on Bengtsson and Yi's definition of the operational semantics [1]. To define the exact state space we need to first introduce the terms *integer valuation* and *clock assignment.*

An integer valuation is of the form $IV : V \rightarrow \text{dom}(V)$, where $\text{dom}(V) = \bigcup_{iv \in V} \text{dom}(iv)$. It maps each integer variable to an integer value of the corresponding domain. With an integer valuation v we use $v \in ig$ to denote that each integer variable in v satisfies the integer guard ig. For an integer valuation v and a set of integer assignments i, the application of i to v assigns each variable x in i a new integer value given by $i(x)$ and retains the other integer variables. The resulting integer valuation is denoted by $[i]v$.

A clock assignment maps each clock variable $x \in C$ to a positive real. With a clock assignment u we use $u \in g$ to denote that each clock variable in u satisfies the guard g. A guard g can either be a clock guard $cg \subseteq B(C)$ or an assigned set of location invariants $I(l)$, where $l \in L$ is a location. The assignment where each clock variable $x \in C$ gets increased by $d \in \mathbb{R}_+$ is denoted by $u + d$. For a clock assignment u and a set of clock resets r, the application of r to u maps all clocks in r to 0 and the other clock variables remain unchanged. The resulting clock assignment is denoted by $[r]u$.

Definition 2.2. (exact state space) Let $T = \langle L, \Sigma, C, V, E, I, l_0 \rangle$ be a timed automaton. The state space of T is defined as a transition system where states are triples $\langle l, v, u \rangle$, with $l \in L$, v is an integer valuation and u is a clock assignment. There are two different transition types:

- $\langle l, v, u \rangle \xrightarrow{d} \langle l, v, u + d \rangle$ if $u \in I(l)$ and $(u + d) \in I(l)$ for $d \in \mathbb{R}_+$
- $\langle l, v, u \rangle \xrightarrow{a} \langle l', v', u' \rangle$ if $l \xrightarrow{cg,ig,a,i,r} l', u \in I(l), u \in cg, v \in ig$, $u' = [r]u$, $v' = [i]v$ and $u' \in I(l')$

One of the most common questions about a timed automaton is the reachability of a given final state. This is a difficult task to solve especially for large systems.

Definition 2.3. (reachability) A state $\langle l, v, u \rangle$ is reachable for a timed automaton if there exists a sequence of transitions from the start state $\langle l_0, v_0, u_0 \rangle$ to a state $\langle l, v, \phi \rangle$ for the constraint $\phi \in B(C)$, when u is satisfying ϕ.

Clock values are real-valued, therefore it may appear that there is an infinite amount of possible reachable states, which could lead to the assumption that the reachability problem is undecidable. With the introduction of *zones* this problem can be solved. Instead of referring to a state as a triple $\langle l, v, u\rangle$ we use zones and define a state as a triple $\langle l, v, z\rangle$. A zone z is a conjunction of constraints that symbolically describes possible values for all clock variables in u. In other words, a zone provides a zone constraint for each clock variable. For example, z could be of the form $(0 \leq x \leq 12 \wedge y == 0)$ and therefore the zone z contains all clock assignments which satisfy these two zone constraints.
A zone z is a subsumption of a zone z' if the clock assignments contained by z are a subset of the clock assignments contained in z'. This is the case if all zone constraints of z' are implied by z.
Instead of delay and action transitions the finite state space called *zone graph* only uses one transition type, which we call *zone transition.* A zone transition is a combination of action and delay transitions applied to a zone graph.

Definition 2.4. (zone graph) Let $T = \langle L, \Sigma, C, V, E, I, l_0\rangle$ be a timed automaton. A zone graph of T is a transition system where states are triples $\langle l, v, z\rangle$, with $l \in L$, v is a integer valuation and z is a zone. A zone transition is defined as:

- $\langle l, v, z\rangle \xrightarrow{a} \langle l', v', z'\rangle$ if $l \xrightarrow{cg,ig,a,i,r} l', z \in cg, v \in ig$, $v' = [i]v$, where
  - each clock x of z gets its lower bound in z' by the smallest possible bound of cg, r and the original lower bound of x in z, and
  - each clock x of z gets its upper bound in z' by $I(l')$.

We distinguish between two main transition types. The zone transitions refer to transitions between states of a zone graph. In contrast, *structural transitions* refer to the transitions induced by the edges of a concurrent system of timed automata. We define the structural transitions of a concurrent system as either *synchronous* or *asynchronous.* A synchronous structural transition consists of two edges from different automata with the same synchronisation labels, whereas an asynchronous structural transition only consists of one edge with a special void label. The synchronisation labels are given in Σ.
For simplicity we use the term transition for a structural transition and otherwise use the explicit term zone transition.[1]

[1] In classical state space search and planning, structural transitions correspond to operators, whereas zone transitions correspond to transitions between states in the state space.

2.2 Greedy Best-First Search

Greedy best-first search (GBFS) is a simple and common search algorithm in planning, used for finding a trace to a goal state in a state space. It builds the algorithmic base of the under-approximation refinement algorithm and is therefore essential. GBFS always tries to expand the most promising node in our state space graph. In our case this is a state with best heuristic value [11].

2.2.1 Algorithm

The GBFS algorithm, as seen in Algorithm 1, uses two lists, one to store already explored states and one to store non-explored states. Since we always want to explore the most promising state, it is effective to implement the open list as a sorted list, for example as a priority queue. The algorithm starts in an initial state and always expands the state with the lowest heuristic value. After a state gets extracted from the open list (line 5) we check if it is a goal state (line 6–7). This ensures that we do not unnecessarily expand states which already satisfy our goal. If the state is not a goal state, we check if we have already explored it (line 8) and if this is not the case, we expand its successors. If a child state is not contained in the closed list and therefore not already explored, we insert it into the open list.

Algorithm 1 Greedy Best-First Search

```
 1: procedure (explore())
 2:     openList = [initialState]
 3:     closedList = [ ]
 4:     while NOT openList.isEmpty() do
 5:         state = openList.popMin()
 6:         if isGoal(state) then
 7:             return trace(state)
 8:         if NOT closedList.contains(state) then
 9:             closedList.insert(state)
10:             for each child $\in$ successors(state) do
11:                 if NOT closedList.contains(state) then
12:                     openList.insert(child)
```

2.2.2 Properties

The GBFS algorithm's properties mainly depend on its heuristic function. If the heuristic is safe and therefore the heuristic only assigns the infinite value to states that are unreachable, the algorithm is complete and ensures to find a solution if it exists. Since we only consider the heuristic value and not the path length the GBFS algorithm is not optimal. With a good heuristic function the GFBS algorithm leads to a solution very fast.

GBFS has a problem when it encounters local minima or plateaus. A local minimum is reached if all successors of an explored state have an heuristic value that is higher compared to the heuristic value of the explored state. Similar to a local minimum, a plateau is reached if all successors of an explored state have the same heuristic value as the explored state itself. In such a case the GBFS algorithm wanders without any real direction to a goal state and therefore loses a lot of performance.

2.3 Directed Model Checking

The goal of a verification tool is to verify the correctness of a system. This task grows of importance since systems are increasing in size and complexity. Model checking can be done manually which is a strenuous task for big systems since the number of states grows exponentially to the size and complexity of the systems. A model checker should return the error trace if a bug is found. With the help of this error trace the bug can be reproduced and fixed.
Directed model checking is an approach to the problem of finding error states. It is used to find short error traces, in other words to find paths to error states as fast as possible. To do so we use a search algorithm and a heuristic distance function which is guided to reachable error states. The heuristic function assigns a value to each state, which reflects a distance estimation to the closest error states. Therefore states with a low heuristic value are preferably explored. In principle this is a heuristic search and therefore we can use algorithms like GBFS and A* [12]. This approach leads to error states quickly but has a downside: the whole state space has to be explored to prove the correctness of a system.
Directed model checking has two parameters which determine its effectiveness. One is the abstraction to compute the distance heuristic and the other one is the search algorithm. This leads to a lot of similarities with planning algorithms in artificial intelligence that use heuristics. [5, 6]

2.3.1 Mcta

Mcta is a verification tool for timed automata [9]. It is optimised to find short traces to error states and therefore to help with bug finding in real-time systems. Mcta has implemented directed model checking algorithms and heuristics, and is easily expandable [3]. To find paths to error states it uses the directed model checking approach. The GBFS algorithm of Mcta is used as a base for the implementation of the under-approximation refinement algorithm.
One of the implemented algorithms in Mcta besides GBFS is the useless transitions algorithm (UT) [6, 13]. UT is a transition-based algorithm that tries to estimate useless transitions with the help of a heuristic function. UT defines a structural

transition t, that induces a zone transition from a state s to its successor state s', as relatively useless if the heuristic value in s in an under-approximated system, where t has been removed, is lower or equal compared to the heuristic value of s' in the original system. If s' is reached by such a relatively useless transition, the algorithm penalises s' by adding a penalty value to its heuristic value, leading to similarities with A*. While UT uses under-approximations adaptively in every state and dependent on the previously applied transition, the overall search is still performed on the original zone graph induced by the problem.
Mcta uses the synchronous and asynchronous transitions to generate a zone graph of a given concurrent system of timed automata on-the-fly. Therefore it is possible to run concurrent systems without having to build the product of the timed automata. This is especially useful since building the product of several automata can be very complex and time consuming due to the state explosion problem.

2.3.2 Duplicates

The duplicate detection in directed model checking used for timed automata does not only detect states that are exactly the same, but also states with subsumptions of zones. For example, the expansion of a zone graph of the timed automaton from Fig. 2.1b is shown in Fig. 2.2. The red marked state represents a subsumption of the state with the location *start* and is therefore detected as a duplicate. This is the case since all the integer valuations and location invariants of the red marked state are identical to the one of the initial state except for the location invariant relating to y. The clock variable y in the red marked state has a lower bound with the value 5 and y in the initial state has a lower bound of 0. This implies that all the values that y can be assigned to in the red marked state are also possible in the initial state. Therefore the zone of the red marked state is a subsumption of the zone given by the initial state.

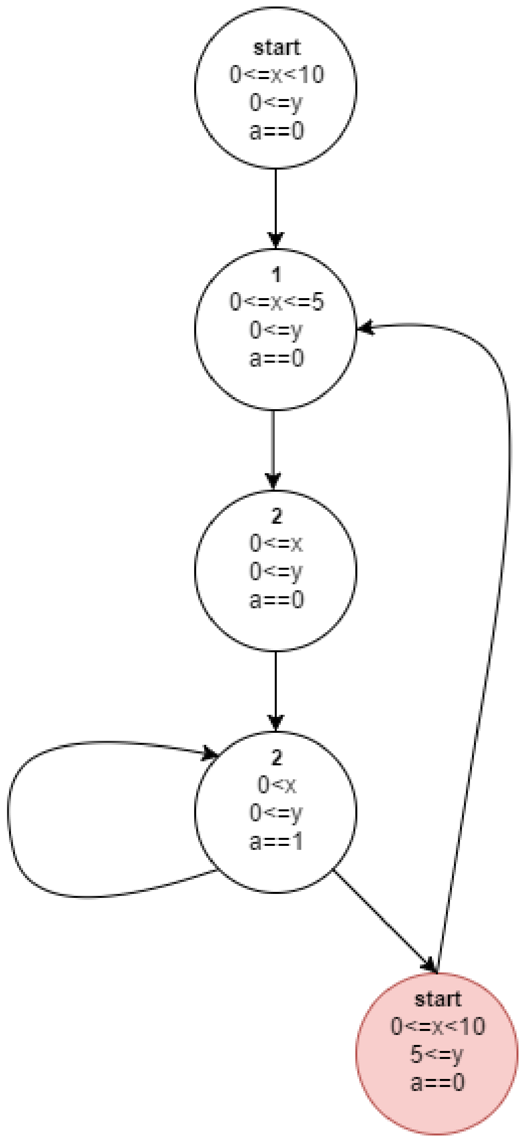


Figure 2.2: Zone graph of the timed automaton from Fig. 2.1b.

3 Under-Approximation Refinement

This chapter covers the idea behind the under-approximation refinement approach and shows how under-approximations are defined for concurrent systems of timed automata. Furthermore, we present a basic under-approximation algorithm which was implemented for the evaluation.

3.1 Idea

One major problem in planning is the state explosion problem [14]. In other words, the size of a state space grows exponentially compared to the planning problem size. A common solution to this problem is to not just evaluate the states but also evaluate transitions and preferably apply transitions that seem to be guided to a solution [8, 13, 15]. This approach is based on the observation that in most planning tasks the amount of used transitions to find a solution is often just a small subset of all available transitions. Table 3.1 shows that the same observation also applies to directed model checking for several timed automata case studies. Although almost 75% of the available transitions are needed for the studies N and M, we observe that the number of needed transitions is very low for the remaining case studies.
The under-approximation refinement approach proposed by Heusner et al. [8] tries to exploit this observation. The under-approximation algorithm allows only a small subset of all transitions to be applicable. If more transitions are needed to find an error state, the so called *refinement guard* is triggered and more transitions are added. Eventually if necessary all transitions are applied and therefore the algorithm stays complete. Since this process only limits the available transitions during the search process, all found solutions are still valid.
Whilst the algorithm was originally designed for planning tasks we adapt it to directed model checking tasks to find error traces in concurrent systems of timed automata by implementing it into Mcta.

	GBFS	**UT Search**
Ø A	12.74%	10.39%
Ø C	16.25%	11.58%
Ø D	11.17%	8.37%
Ø N	74.41%	71.64%
Ø M	75.63%	72.85%

Table 3.1: Overview of the percentage of used transitions in a solution for five different case studies called A, C, D, N and M. Transitions are denoted by the synchronous and asynchronous transitions of the concurrent system of timed automata. Both, the greedy best-first search (GBFS) and the useless transitions (UT) algorithm are shown. UT is a transition-based directed model checking algorithm implemented in Mcta.

3.2 Definition

An under-approximation of a timed automaton $A = \langle L, \Sigma, C, V, E, I, l_0 \rangle$ is a timed automaton $UA = \langle L, \Sigma, C, V, E', I, l_0 \rangle$, where E' is a subset of E. The error trace of an under-approximation is always an error trace of the original timed automaton since an under-approximation is at least as strict as the original timed automaton. This implies that the zone graph of an under-approximation only consists of the states and transitions of the zone graph of the original timed automaton and is therefore always lesser or equal in size. If a state is not reachable in an under-approximation UA, we cannot infer that the same state is also not reachable in A. Therefore a validation of a timed automaton cannot be done on an under-approximation of the timed automaton.

An under-approximation of a concurrent system of timed automata $M = \{A_1, A_2, ..., A_n\}$, where each A_i with $i \in n$ denotes a timed automaton, is a concurrent system of timed automata $UM = \{UA_1, UA_2, ..., UA_n\}$, where each UA_i with $i \in n$ denotes an under-approximation of a timed automaton A_i. A zone graph of an under-approximation of a concurrent system of timed automata has the same properties as a zone graph of an under-approximation of timed automata.

3.3 Algorithm

It is possible to implement the idea behind the under-approximation refinement algorithm in many different ways. There are three basic components for an under-approximation refinement algorithm: a search algorithm, a *refinement guard* and a *refinement strategy*. The search algorithm expands the zone graph of an under-approximation of a concurrent system. The refinement guard determines when a refinement step has to be done. Lastly, the refinement strategy evaluates and adds transitions to the transition set of the last under-approximation therefore creating a new under-approximation.

The following Algorithm 2 is easy to implement and uses greedy best-first search as its base search algorithm. The refinement strategy utilises relaxed plans to find possibly useful transitions.
The initially allowed transitions are given by making a relaxed plan of the initial state and allowing all used transitions in the trace of the relaxed plan (line 4). The refinement guard is triggered if the algorithm reaches a plateau or local minimum. This is the case if no states with a lower heuristic value are inserted into the open list. It is also triggered if the open list gets empty, this is needed such that the algorithm stays complete (line 14).
As a refinement strategy the algorithm makes a relaxed plan of each state in the closed list with a minimal heuristic value and allowing the transitions used in the trace of each relaxed solution (line 20–24). If no new transitions are found in such a refinement step, we continue by making relaxed plans of each state in the closed list with a minimal heuristic value plus 1 (line 28–29). This gets repeated until a refined transition subset is found or the closed list is scanned completely.
If a refined transition subset is found, we reopen all states of the closed list which have a newly allowed transition (line 25–27). If no new transitions are allowed and the open list is not empty, we continue the search without a refined subset.
If the open list is empty and no new transitions were found, we restart the refinement process (line 16–17) but this time adding all applicable transitions instead of only adding transitions used in relaxed plans (line 31–40). This also ensures that in the worst case we add all transitions to the subset and the algorithm hence stays complete.
Each refinement step only adds more transitions, therefore the already expanded zone graph can be further used for a less strict under-approximation and is not needed to be expanded again.
The use of relaxed plans in the refinement strategy is based on the assumption that the traces of relaxed plans contain mostly transitions which also lead to an error state in the real zone graph.

Algorithm 2 Basic implementation of the under-approximation refinement idea using GBFS as a base algorithm.

```
procedure (explore())
    openList = [ ]
    closedList = [initialState]
    refineOperators(initialState.heurValue())
    while NOT openList.isEmpty() do
        state = openList.popMin()
        if isGoal(state) then
            return trace(state)
        if NOT closedList.contains(state) then
            closedList.insert(state)
            for each child ∈ successors(state) do
                if NOT closedList.contains(state) then
                    openList.insert(child)
        if openList.IsEmpty() OR NOT openList.betterStateInserted() then
            refineOperators(closed.minHeurValue())
        if openList.IsEmpty() then
            addApplicableOperators(closed.minHeurValue())

procedure (refineOperators(heurValue))
    newOperators = FALSE
    stateSubset = openList.getStates(heurValue)
    for each state ∈ stateSubset do
        relaxedPlan = relaxedPlan(state)
        if addNewOperators(relaxedPlan) then
            openList.insert(state)
            closedList.remove(state)
            newOperators = TRUE
    if NOT newOperators AND heurValue != closedList.maxHeurValue() then
        refineOperators(heurValue++)

procedure (addApplicableOperators(heurValue))
    newOperators = FALSE
    stateSubset = openList.getStates(heurValue)
    for each state ∈ stateSubset do
        if addNewOperators(state) then
            openList.insert(state)
            closedList.remove(state)
            newOperators = TRUE
    if NOT newOperators AND heurValue != closedList.maxHeurValue() then
        addApplicableOperators(heurValue++)
```

3.4 Implementation Details

For the implementation of the greedy best-first search algorithm usually a priority queue is used as an open list and a hash list is used for the closed list. This has the advantage that inserting duplicates into the closed list can be easily avoided by calculating the hash value of a state and comparing it with the hash values in the closed list. Thus no iteration of the closed list is needed, since the hash values can be stored in an order.

For the under-approximation algorithm proposed in the previous section we have to be able to reinsert states of the closed list into the open list and therefore also to remove states from the closed list. Doing this efficiently with a hash list is a difficult task since we only store the hash values and not the states themselves.

To have the efficiency of a hash list and still being able to get states back out of the list, we introduce a second priority queue "notComputed". It stores all generated states that have not been used yet in a refinement step to allow the transitions of their relaxed plans. For the refinement we now only have to iterate through the "notComputed"list, create relaxed plans of the states with the desired heuristic value and remove those from the "notComputed"list. This "notComputed"list allows us to create relaxed plans of states in the closed list and also has the benefit that we do not need to iterate through the closed list to search states with a desired heuristic value.

As a last step of the refinement we need to reinsert all the states of the closed list which have a newly applicable transition available into the open list. Since we cannot do this with our hash list, we introduce a new list "permittedStates". It contains all the states which were successors of explored states from the open list but were not yet allowed, since the generating transition was not part of the allowed transition subset. Instead of reinserting states of the closed list into the open list, we now iterate at the end of each refinement through the "permittedStates"list and reinsert those states into the open list, which have a newly allowed transition from its predecessor to the state itself.

4 Evaluation

To evaluate the under-approximation refinement algorithm (UA) we implemented it into the Mcta tool [3]. It is compared to the already implemented algorithms greedy best-first search (GBFS) and useless transitions (UT) [6, 13] with the provided benchmark set of Mcta's website [16]. The comparison with GBFS shows the differences to a simple algorithm and the comparison with UT leads to interesting insights since UT is a fast transition-based algorithm.

4.1 Setup

The evaluation system is a 64-bit virtual machine running Ubuntu with a 4GB memory bound, 6 logical processing cores of an Intel Core i7-4770k CPU with 3.5GHz and a timeout of 30 minutes. All tests run on the same system. To reduce noise each test is conducted three times and the shown test results are the logs of the tests with the median in time.

The number of explored states of the under-approximation refinement algorithm is given by the states which have been extracted of the open list. It is not taken into account if such an extracted state has any allowed successors. Each state is counted at most once as explored.

The algorithms are tested on six different test sets which each contain several tests themselves. The test sets C, D and E are all from the same industrial case study, where a real-time controller system of tram tracks was examined. Test set E only contains systems without any reachable error states. The set D is a harder version of the systems in test set C. The sets M and N model a real-time communication protocol with the test set N being a harder version of M. Test set A is a toy set which models arbiter trees.

The results were obtained using the command line option $-c2$ which enables cashing for the discrete part of states. This option leads to better runtimes since the heuristic values of similar states do not have to be recomputed.

4.2 Results for the h^U Heuristic

To evaluate the UA algorithm, we compare it to GBFS and UT. In this section we use the h^U heuristic which is the most informed heuristic of Mcta that the current implementation of UA supports. Therefore using the h^U heuristic should show the best performing results. The h^U heuristic is an adaptation of the FF heuristic for timed automata and based on relaxed plans which has the downside of a lot of computational effort to compute the heuristic values of states [4].

4.2.1 Error Traces

We compare the ability to find error traces by testing the three algorithms (GBFS, UT and UA) on the test sets A, C, D, N and M, which all contain systems with reachable error states.

Table 4.1 leads to the observation that UA almost always returns an error trace while not exceeding the memory bound, with the exception being test D9. The runtimes of UA are reproducable and overall seem to be competitive compared to the runtimes of UT. Only the test set D leads to some inconsistencies where the runtime does not seem to be dependent on the difficulty of the problems.

The comparison with GBFS shows that in almost every test UA leads to a better runtime, only the two tests D7 and D8 are completed faster by the GBFS algorithm. In those tests GBFS also outperforms UT which illustrates that GBFS can deliver good results if the heuristic values lead to a relatively straight path to an error state without many plateaus or local minima.

The runtimes of UT are similar to ones of UA, although in most tests UA leads to even better runtimes than UT. This can seem a bit strange since UA almost always explores far more states than UT but can be explained with a less time intensive computation of useful transitions of UA compared to UT. Both algorithms have some difficulties to find error states in the test set D, where they have inconsistent runtimes compared to the difficulty of the tests. This might be due to the GBFS base algorithm which both UA and UT use and also due to difficulties in finding transitions that lead to a close and reachable error state.

The used memory and the number of explored states always have a strong resemblance since most of the memory is used for storing states. The memory consumption of UA is similar to the memory consumption of UT since both restrain the usage of transitions and therefore the zone graph is kept rather small. The fact that UA is most of the time exploring far more states than UT and still manages to keep a similar memory consumption shows that UT needs to store more data than UA. The used memory of UA is almost in all tests far less than the one of GBFS since UA finds an error state with much less explored states.

The trace lengths of UA are in most of the tests a bit longer than the ones of UT but in all tests show an improvement over the trace lengths of GBFS.

	runtime in s			**used memory in MB**			**explored states**			**trace length**		
	GBFS	**UT**	**UA**	**GBFS**	**UT**	**UA**	**GBFS**	**UT**	**UA**	**GBFS**	**UT**	**UA**
A2	**0.0**	**0.0**	**0.0**	**60**	**60**	**60**	25	**20**	**20**	21	**18**	**18**
A3	**0.0**	0.01	**0.0**	**61**	**61**	**61**	82	**27**	46	18	**17**	**17**
A4	**0.01**	0.04	**0.01**	**62**	**62**	**62**	39	**34**	131	28	**22**	23
A5	0.48	0.17	**0.08**	72	**68**	72	4027	**42**	586	47	**27**	29
A6	-	**1.0**	1.89	-	**91**	254	-	**50**	5564	-	**32**	35
C1	**0.01**	**0.01**	**0.01**	**61**	**61**	**61**	429	**243**	239	67	**54**	55
C2	**0.01**	0.02	**0.01**	**61**	**61**	**61**	828	**212**	239	83	**54**	55
C3	**0.01**	0.02	**0.01**	**61**	**61**	**61**	1033	**198**	239	79	**54**	55
C4	0.15	**0.02**	0.03	64	**61**	**61**	12k	**174**	1117	112	**55**	64
C5	0.86	**0.03**	0.04	78	**61**	**61**	65k	**147**	1493	176	**61**	75
C6	5.46	**0.03**	0.05	166	**61**	62	453k	**147**	1493	432	**61**	75
C7	45.42	**0.04**	0.05	974	**61**	62	4230k	**143**	1493	924	**61**	75
C8	31.46	0.32	**0.09**	758	**62**	63	3403k	**1466**	2875	2221	**56**	161
C9	-	0.36	**0.2**	-	**62**	66	-	**1575**	6119	-	**69**	169
D1	0.05	0.11	**0.02**	**61**	**61**	**61**	1344	939	**292**	96	**88**	89
D2	3.03	**0.15**	0.65	98	**62**	67	112k	**843**	14k	220	**89**	203
D3	0.91	0.13	**0.06**	75	**62**	**62**	44k	**717**	1228	241	**89**	95
D4	6.36	**0.15**	3.31	138	**62**	94	259k	**615**	83k	410	**89**	262
D5	0.22	11.76	**0.06**	64	125	**62**	4455	87k	**720**	115	**107**	108
D6	**0.52**	-	0.59	**67**	-	**67**	12k	-	**7490**	301	-	**206**
D7	**0.82**	4.8	2.49	**70**	80	85	20k	**18k**	34k	154	**109**	128
D8	1.01	**0.63**	121.53	72	**64**	1186	23k	**1883**	1808k	259	**109**	253
D9	-	**0.59**	-	-	**64**	-	-	**1533**	-	-	**110**	-
M1	0.02	0.06	**0.01**	**61**	**61**	**61**	7668	4366	**1529**	71	73	**66**
M2	0.06	0.05	**0.02**	63	**61**	**61**	18k	**2018**	3852	119	**81**	105
M3	0.06	0.39	**0.02**	63	64	**61**	19k	17k	**4794**	124	163	**93**
M4	0.14	0.5	**0.05**	68	66	**63**	46k	15k	**12k**	160	**91**	148
N1	0.05	0.09	**0.02**	62	62	**61**	9117	5191	**1880**	99	80	**68**
N2	0.14	0.09	**0.06**	66	**62**	63	23k	**3260**	8106	154	136	**103**
N3	0.27	0.41	**0.06**	69	65	**63**	43k	19k	**7117**	147	149	**87**
N4	1.08	0.46	**0.19**	88	**67**	**67**	152k	**15k**	25k	314	377	**185**

Table 4.1: Overview of runtime, used memory, explored states and trace length of the test sets which contain error states. A "-" indicates that the test ran out of memory. The best result per test is presented in bold.

4.2.2 Allowed Transitions

It is important for the UA algorithm to find useful transitions early and efficiently. In the best case the algorithm only allows transitions that are used in the trace of an error state to avoid exploring states which make no progress to an error state. Table 4.2 shows that UA allows only around a third of all transition in almost all of the tests. This obviously leads to a smaller explored zone graph than the one of GBFS. In the test sets N and M the UA algorithm allows significantly more transitions than in the other test sets. This is caused by the fact that the test sets N and M need over 50% of all transitions in the error trace of a state. This is not only the case for the UA algorithm but also for GBFS and UT as shown in Table 3.1.

	allowed	used	all	allowed in %	used in %
A2	21	15	73	28.77%	20.55%
A3	32	17	161	19.88%	10.56%
A4	43	23	337	12.76%	6.82%
A5	55	29	689	7.98%	4.21%
A6	82	35	1393	5.89%	2.51%
C1	80	30	240	33.33%	12.5%
C2	80	30	242	33.06%	12.4%
C3	80	30	242	33.06%	12.4%
C4	110	32	278	39.57%	11.51%
C5	92	34	314	29.3%	10.83%
C6	92	34	322	28.57%	10.56%
C7	92	34	330	27.88%	10.3%
C8	124	44	340	36.47%	12.94%
C9	128	46	358	35.75%	12.85%
D1	104	41	478	21.76%	8.58%
D2	220	53	490	44.9%	10.82%
D3	146	44	490	29.8%	8.98%
D4	244	59	500	48.8%	11.8%
D5	128	49	724	17.68%	6.77%
D6	218	65	730	29.86%	8.9%
D7	218	56	736	29.62%	7.61%
D8	356	72	746	47.72%	9.65%
M1	27	24	41	65.85%	58.54%
M2	31	26	44	70.45%	59.09%
M3	30	27	44	68.18%	61.36%
M4	34	29	47	72.34%	61.7%
N1	27	24	41	65.85%	58.54%
N2	32	26	44	72.73%	59.09%
N3	30	27	44	68.18%	61.36%
N4	34	29	47	72.34%	61.7%
Ø	96.42	34.94	340.77	41.88%	25.98%

Table 4.2: This table shows an overiew of the number of allowed transitions from the under-approximation refinement algorithm. It also presents the number of used transition in the trace of the found error state.

The sets N and M illustrate that UA is still a viable option if a larger amount of transitions is needed to find an error state. It is possible to say that UA outperforms UT in the test sets N and M because a lot of transitions are needed. But this conclusion can not clearly be drawn since both test sets are from the same case study and do not deliver enough diverse systems for a reliable conclusion. Fig. 4.1 illustrates that the UA algorithm is able to find almost only useful transition in the test M4. UA is able to add most of the needed transitions for the error trace in the first few refinement steps, which is not only typical for the M and N test sets but

also for almost all tests in which UA performed really well.
In the tests D2, D4 and D8 the UA algorithm allows over 44% of all transitions. This explains why the UA algorithm has unusually big runtimes in those tests compared to their intended difficulty. This is due to a difficulty in finding useful transitions with relaxed plans as can be seen in Fig. 4.2. Most of the added transitions during the refinement steps are not used in the trace of the found error state.

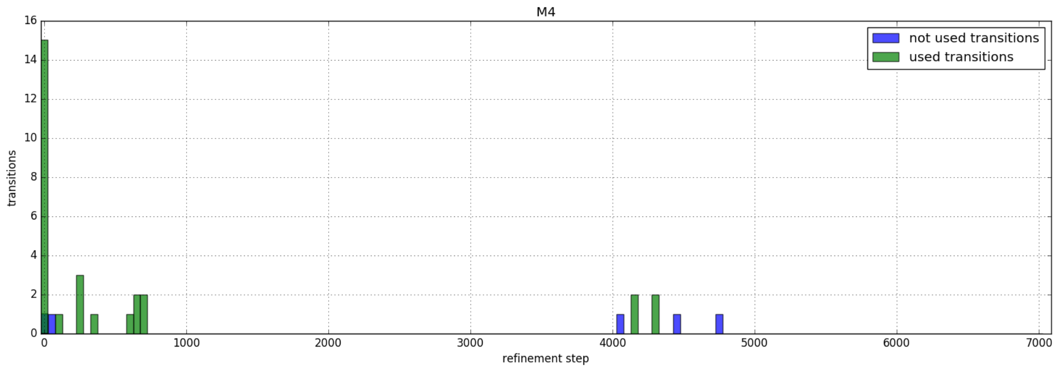


Figure 4.1: This diagram shows at which refinement step how many transitions were added in the test M4 and if they were used in the trace of the found error state. The transitions are combined in bins of the size 16 to make a visualization possible. Therefore each bar represents the added transitions of 16 refinement steps.

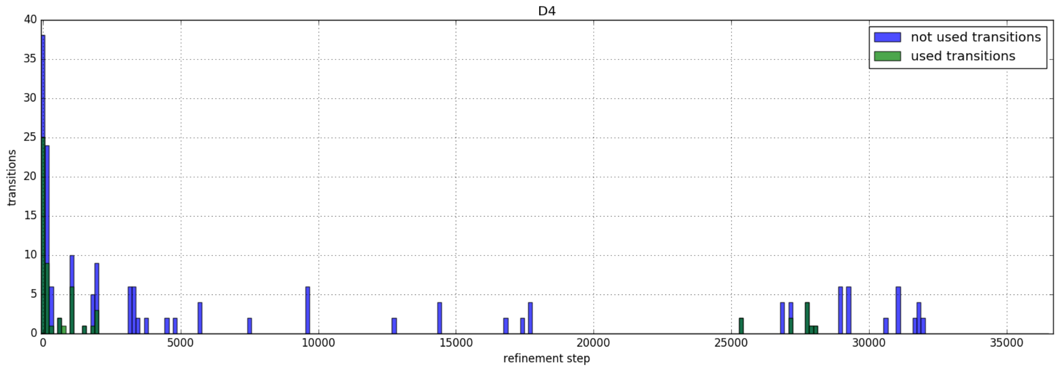


Figure 4.2: This diagram shows at which refinement step how many transitions were added in the test D4 and if they were used in the trace of the found error state. The transitions are combined in bins of the size 150 to make a visualization possible. Therefore each bar represents the added transitions of 150 refinement steps.

4.2.3 Heuristic Curves

To analyse the heuristic behaviour of the algorithms we examine their heuristic curves. Fig. 4.3 shows a typical heuristic graph for all tested algorithms. We observe that the GBFS algorithm can easily get stuck at a plateau and has some difficulty to find a better state. UT finds a solution without expanding a lot of states but

UA is still able to outperform UT in terms of runtime since the calculation of useful transitions is faster. In most tests this is the reason why UA has good runtimes even though the algorithm expands more than 10 times more states than UT.

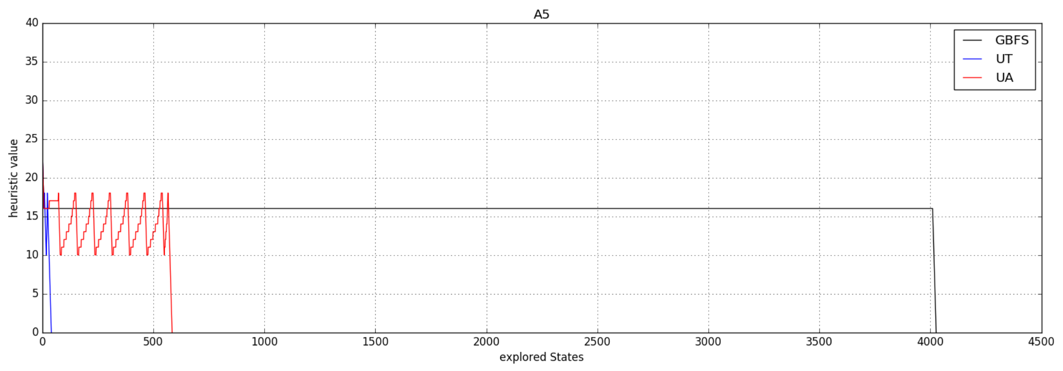


Figure 4.3: A graph that shows the heuristic value of each explored state in a time-line. The results of the algorithms GBFS, UT and UA are plotted for the test A5.

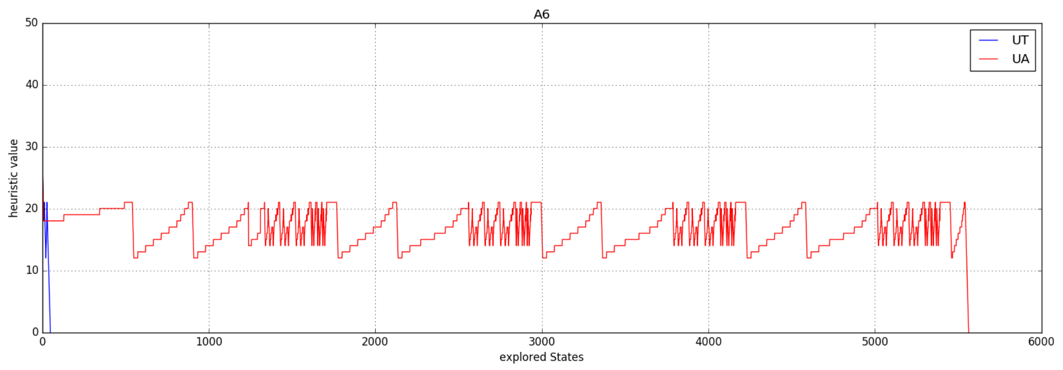


Figure 4.4: A graph that shows the heuristic value of each explored state in a time-line. Only the algorithms UT and UA are plotted for the test A6. In this test GBFS was not able to find an error state in the given memory bound of 4GB.

In Fig. 4.4 we plot a typical graph of a test where UT performs better than UA. UT again finds a solution very quickly while UA gets stuck in a local minimum and needs repeated refinement steps until a way out of the local minimum is found. This shows that the UA algorithm can still have a problem with plateaus and local minima if the relaxed plans use similar transitions or even the same transitions that are already allowed. Therefore it is possible that no useful transitions which lead out of an plateau are computed and added to the transition set. This observation causes the assumption that UA could benefit from a different refinement strategy, for example a strategy similar to the estimation of useless transitions in UT.

4.2.4 Verification

To check the ability of using the UA algorithm as a verification tool, we test the algorithm with the set E. This set of tests contains timed automata which have no reachable error state. In order to conclude that a timed automaton has no error states, a algorithm needs to explore the whole state space.

	runtime in s			**used memory in MB**			**explored states**		
	GBFS	**UT**	**UA**	**GBFS**	**UT**	**UA**	**GBFS**	**UT**	**UA**
E1	0.2	0.67	0.39	65	66	66	17k	17k	17k
E2	86.3	296.55	205.9	1409	1879	1567	4659k	4675k	4206k
E3	-	-	-	-	-	-	-	-	-
E4	-	-	-	-	-	-	-	-	-

Table 4.3: Overview of runtime, used memory and explored states of the test set E which only contains systems without any error states.

The test results in Table 4.3 show that neither of the tested algorithms is optimised for verification purposes. All the algorithms take a lot of time and especially memory to explore the whole state space and are only able to solve the tests E1 and E2 without running out of memory. GBFS shows the best results in both used memory and runtime. This is due to the fact that the transition-based search methods UT and UA use a lot of computational time to calculate the allowed transitions. They both also need more memory than the GBFS algorithm since they have to store data for the calculation of useful transition. UA shows a slight improvement over UT in the runtime which is probably due to a faster way of calculating useful transitions. UA also needs less memory than UT since the number of explored states is smaller for the UA algorithm.

It is important to note that all algorithms explore the whole state space in the tests E1 and E2. The difference in the number of explored and generated states is due to the zone abstraction. If we have two states $s = \langle l, v, z\rangle$ and $s' = \langle l, v, z'\rangle$, where z is a subsumption of the zone z' and we encounter s before s', the state s is explored. A later encounter of s' would also lead to an exploration of s'. But in the case that s' is encountered before s, we only explore s' since s represents a subset of the states of the exact state space represented by s'.

4.3 Results for the $null$ Heuristic

The *null* heuristic assigns the value 0 to each state. This alters the UA algorithm because each encountered state is now part of a plateau. Therefore the UA algorithm always makes a refinement step after exploring each state and allows the transitions of the relaxed plans of every explored state.

When the GBFS algorithm is used with the *null* heuristic it basically resembles a blind search. Unfortunately, the UT algorithm does not support the *null* heuristic since it is not possible for UT to find useful transitions without a guided heuristic. It is important to note that UA has an unfair advantage over GBFS in this comparison since UA still uses the h^U heuristic to make relaxed plans of states during the refinement steps. We still provide this comparison to investigate the pruning power of UA when no further heuristic guidance for the states is used.

	runtime in s		**used memory in MB**		**explored states**		**trace length**	
	GBFS	**UA**	**GBFS**	**UA**	**GBFS**	**UA**	**GBFS**	**UA**
A2	0.0	0.0	60	60	115	24	102	22
A3	0.02	0.01	64	61	13k	639	2046	266
A4	-	0.6	-	102	-	29k	-	1380
A5	-	1075.44	-	3630	-	792k	-	46123
A6	-	-	-	-	-	-	-	-
C1	0.02	0.01	63	61	12k	637	922	465
C2	0.06	0.01	67	61	38k	637	1386	465
C3	0.09	0.01	70	61	54k	637	1524	465
C4	0.97	0.03	146	62	519k	1752	7725	1033
C5	10.05	0.07	786	64	4763k	4820	25647	2490
C6	-	0.09	-	65	-	4820	-	2490
C7	-	0.09	-	67	-	4820	-	2490
C8	-	0.04	-	63	-	2189	-	542
C9	-	0.07	-	65	-	4827	-	541
D1	7.23	0.1	497	64	2493k	11k	4981	9719
D2	-	0.21	-	67	-	21k	-	431
D3	12.56	0.23	836	68	4337k	13k	494	10083
D4	-	7.82	-	260	-	754k	-	647
D5	-	0.51	-	72	-	27k	-	22022
D6	-	1.38	-	88	-	55k	-	44673
D7	-	20.58	-	354	-	553k	-	48759
D8	-	106.71	-	1640	-	3573k	-	833
D9	-	-	-	-	-	-	-	-
M1	0.04	0.02	62	61	12k	4700	2787	1094
M2	0.21	0.04	67	62	50k	10k	13709	3034
M3	0.29	0.09	68	63	72k	25k	11524	2192
M4	1.05	0.43	92	74	235k	115k	52634	11977
N1	0.12	0.05	64	62	15k	6599	3576	804
N2	0.93	0.11	74	64	102k	16k	15894	2541
N3	1.12	0.45	76	69	107k	52k	20086	2778
N4	10.38	3.81	138	100	743k	357k	86236	22863

Table 4.4: Overview of all test sets which contain error states, tested with both the UA and GBFS algorithm using the *null* heuristic.

Table 4.4 shows that UA outperforms GBFS in every test. Both in used memory and runtime UA has always the better values, since the restriction in transitions leads to a smaller zone graph. The trace lengths of UA and GBFS are both big especially compared to the ones with the h^U heuristic. The tests show that even with an unguided search method UA is still able to deliver almost the same runtimes as with the h^U heuristic. In the test D8 the test of UA with the *null* heuristic can even outperform UA with the h^U heuristic as a search heuristic. This is probably due to the fact that the calculation of the null heuristic does not need any considerable time whilst the h^U heuristic is a relatively runtime consuming heuristic to calculate. Therefore the exploration of states is a lot faster with the *null* heuristic.

4.4 Results for the d^U Heuristic

To verify the performance of UA with a less informed heuristic we test all algorithms with the d^U heuristic, also called graph distance sum heuristic. It is based on local graph distances in the timed automata system and is faster to calculate than the h^U heuristic.

	runtime in s			**used memory in MB**			**explored states**			**trace length**		
	GBFS	**UT**	**UA**	**GBFS**	**UT**	**UA**	**GBFS**	**UT**	**UA**	**GBFS**	**UT**	**UA**
A2	0.0	0.0	0.0	60	60	60	23	24	21	13	13	13
A3	0.0	0.0	0.0	61	61	61	296	297	53	39	39	24
A4	0.06	0.17	0.01	68	74	62	19k	19k	121	129	129	35
A5	-	-	0.03	-	-	69	-	-	194	-	-	48
A6	-	-	0.09	-	-	96	-	-	283	-	-	63
C1	0.02	0.06	0.02	62	63	61	11k	9796	1177	823	842	785
C2	0.06	0.16	0.01	67	68	61	33k	31k	1177	1229	1105	785
C3	0.11	0.25	0.02	70	72	61	51k	52k	1177	1032	1144	785
C4	0.96	2.49	0.04	138	159	62	465k	504k	2029	3132	5364	915
C5	10.33	23.38	0.09	765	865	64	4617k	4617k	4960	14034	14000	1931
C6	-	-	0.1	-	-	65	-	-	4960	-	-	1931
C7	-	-	0.1	-	-	66	-	-	4960	-	-	1931
C8	-	-	0.06	-	-	64	-	-	3198	-	-	525
C9	-	-	0.14	-	-	67	-	-	7618	-	-	681
D1	7.18	14.36	0.11	499	720	64	2558k	2337k	11k	1908	8198	9719
D2	56.58	-	0.24	3708	-	68	17807k	-	21k	15366	-	431
D3	11.63	18.26	0.26	841	774	68	4439k	2989k	13k	494	502	10083
D4	-	-	8.77	-	-	272	-	-	728k	-	-	647
D5	-	-	0.57	-	-	73	-	-	27k	-	-	22022
D6	-	-	3.66	-	-	117	-	-	106k	-	-	43571
D7	-	-	36.51	-	-	528	-	-	849k	-	-	41852
D8	-	-	118.47	-	-	1760	-	-	3573k	-	-	833
D9	-	-	-	-	-	-	-	-	-	-	-	-
M1	2.14	0.04	0.02	101	61	61	185k	7557	3629	106224	923	311
M2	0.23	7.48	0.07	69	95	63	56k	294k	19k	13952	51541	310
M3	121.72	0.15	0.08	223	64	63	869k	26k	23k	337857	1280	993
M4	38.47	0.53	0.43	207	71	71	726k	100k	126k	290937	4436	855
N1	0.08	0.21	0.03	63	65	62	10k	19k	4386	2669	1855	260
N2	92.66	1.37	0.22	313	75	65	642k	96k	29k	415585	8986	205
N3	966.33	0.34	111.75	393	66	203	1155k	29k	478k	262642	784	93206
N4	4.23	4.2	2.49	114	88	84	330k	239k	239k	51642	1969	537

Table 4.5: Overview of all test sets which contain error states, tested with the UA, UT and GBFS algorithm using the graph distance sum heuristic also called d^U heuristic.

The table 4.5 shows that both UT and GBFS have a lot of problems finding an error state in the given memory bound in most tests since the d^U heuristic leads

to a relatively inaccurate prediction of the distance to an error state. Because UT calculates the usefulness of transitions with the search heuristic, the algorithm is mislead and rates useless transitions as useful, hence achieving in most tests even worse runtimes than GBFS.
UA is almost unaffected by the not so informed heuristic since it still uses the h^U heuristic to calculate relaxed plans for the allowed transitions. Therefore UA obviously has an unfair advantage over UT and GBFS. The performance in the test set A is very surprising since UA with the d^U heuristic performs even better than UA with the h^U heuristic. This is probably caused by finding useful transitions earlier due to a bit of luck with the states used in the refinement steps.

4.5 Plateau Guard

The tested UA algorithm is a basic algorithm which shows the potential of an under-approximation for directed model checking. For the UA algorithm the two most important factors are the refinement guard and the refinement strategy. The previous tests have shown that relaxed plans often lead to useful transitions without allowing too many useless transitions.

	# of refinements
Ø A	1076
Ø C	938
Ø D	97934
Ø M	3493
Ø N	7273

Table 4.6: Shows the average times a refinement guard was triggered for each test set using the h^U heuristic. The tests that run out of memory or time are not used to calculate the average.

Table 4.6 shows that the proposed refinement guard of encountering a plateau or local minimum is triggered very often especially in the more difficult test sets. A possible easy way to improve the UA algorithm is introducing a *plateau guard* to the refinement guard and therefore not refining the transition set every time when a plateau or local minimum is encountered but only after a certain repeated encounters of states which have no better heuristic value. The idea behind this is that small local minima, where just a few states have worse heuristic values, should not lead to a refined transition set.
To test this theory we implemented a plateau guard option and tested it with different values. Table 4.7 shows the average times a refinement guard with a plateau guard was triggered in a test set. We can observe that the number of refinements almost always gets smaller with an increasing plateau guard but fewer refinements

do not have to lead to a faster runtime.
Table 4.8 shows that a plateau guard can have positive effects on the runtime. In every average time of the test sets the best time is achieved by a UA variant that uses a plateau guard. Interestingly, large plateau guard values have an especially positive effect on the test set D where the basic UA algorithm has problems. A plateau guard bigger than 1000 even leads to an error trace in the test D9 where the basic UA algorithm runs out of memory. But the test sets C, M and N show that a large plateau guard can also lead to worse runtimes than the original algorithm. It is important to note that tests like D8, where the runtime is especially large without a plateau guard, have a big impact on the average runtime if such a test can be solved quickly.

	# of refinements												
Guard	**0**	**5**	**10**	**25**	**50**	**100**	**150**	**500**	**1k**	**1.5k**	**2k**	**2.5k**	**5k**
ØA	1076	182	100	43	23	13	9	4	3	2	2	2	2
ØC	938	151	83	26	16	11	10	9	9	9	9	9	10
ØD	97934	16392	8801	3183	77	30	15	14	13	14	14	14	14
ØM	3493	643	339	146	67	43	27	14	12	12	12	11	11
ØN	7273	1250	674	283	137	65	48	20	15	13	12	11	11

Table 4.7: Comparison of the average number of refinement steps until an error state was found per test set from the UA algorithm with different plateau guards. In the shown tests the h^U heuristic was used.

	runtime in s												
Guard	**0**	**5**	**10**	**25**	**50**	**100**	**150**	**500**	**1k**	**1.5k**	**2k**	**2.5k**	**5k**
ØA	0.4	0.35	0.37	0.39	0.35	0.36	0.37	0.4	0.37	0.33	0.28	**0.25**	0.47
ØC	0.05	0.05	0.05	**0.04**	**0.04**	0.05	0.06	0.1	0.1	0.14	0.13	0.14	0.18
ØD	16.09	16.47	15.62	13.81	0.52	0.42	**0.21**	0.32	0.34	0.54	0.46	0.78	0.6
ØM	0.03	0.03	0.03	0.03	**0.02**	0.03	0.03	0.04	0.04	0.04	0.05	0.05	0.06
ØN	0.08	0.07	0.08	0.08	0.07	**0.06**	0.07	0.08	0.09	0.1	0.12	0.14	0.19
D8	121.53	124.97	120.03	106.4	**0.06**	0.37	0.13	0.3	0.46	0.72	0.73	1.98	0.9
D9	-	-	-	-	-	-	-	-	**2.18**	3.63	18.61	5.69	9.46

Table 4.8: Comparison of average runtimes of each test set resulting from the UA algorithm with different plateau guards and the h^U heuristic. The best result per test is presented in bold. Also the tests D8 and D9 are attached since they show some interesting data.

The shown results illustrate that a plateau guard can have both positive and negative effects on the runtime. This simple approach of a plateau guard does not lead to an algorithm that achieves significant better values in every single test. The main conclusion is that it is possible to further improve the proposed UA algorithm with minor changes. A more sophisticated procedure for a plateau guard could probably outperform the basic algorithm in every test but since there are just a few benchmarks available, the probability to over-fit the algorithm to the available tasks is high.

5
Conclusion

We have adapted and implemented the under-approximation refinement algorithm by Heusner et al. [8] for directed model checking in concurrent systems of timed automata. In addition we proposed a simple improvement to the algorithm by introducing a different refinement guard and tested the approach with different parameters.

The evaluation shows that under-approximation refinement is a fast and memory efficient algorithm that can compete in directed model checking. It clearly outperforms the greedy best-first search algorithm and is able to achieve a similar performance as the useless transition algorithm implemented in Mcta.

The proposed addition of a plateau guard shows that the under-approximation refinement algorithm can be improved with some minor changes. With different approaches to the refinement guard and different refinement strategies further improvements are imaginable. Even a combination of the useless transitions algorithm with the under-approximation refinement algorithm, where the allowed transitions of an under-approximation are further evaluated by the useless transitions approach, is possible.

Additionally an evaluation of the algorithm with a range of more diverse benchmark sets could lead to more conclusive insights into the behaviour of the algorithm in directed model checking.

Bibliography

[1] Bengtsson, J. and Yi, W. Timed Automata: Semantics, Algorithms and Tools. In *Lecture Notes on Concurrency and Petri Nets: LNCS 3098*, pages 87–124. Springer-Verlag (2004).

[2] Dräger, K., Finkbeiner, B., and Podelski, A. Directed Model Checking with Distance-Preserving Abstractions. In *SPIN*, pages 19–34 (2006).

[3] Wehrle, M. and Kupferschmid, S. Mcta: Heuristics and Search for Timed Systems. In *FORMATS*, pages 252–266 (2012).

[4] Kupferschmid, S., Hoffmann, J., Dierks, H., and Behrmann, G. Adapting an AI Planning Heuristic for Directed Model Checking. In *SPIN*, pages 35–51 (2006).

[5] Wehrle, M. and Kupferschmid, S. Context-Enhanced Directed Model Checking. In *SPIN*, pages 88–105 (2010).

[6] Wehrle, M., Kupferschmid, S., and Podelski, A. Transition-based Directed Model Checking. In *TACAS*, pages 186–200 (2009).

[7] Richter, S. and Helmert, M. Preferred Operators and Deferred Evaluation in Satisficing Planning. In *ICAPS*, pages 273–280 (2009).

[8] Heusner, M., Wehrle, M., Pommerening, F., and Helmert, M. Under-Approximation Refinement for Classical Planning. In *ICAPS*, pages 365–369 (2014).

[9] Kupferschmid, S., Wehrle, M., Nebel, B., and Podelski, A. Faster than UPPAAL? In *CAV*, pages 552–555 (2008).

[10] Alur, R. and Dill, D. L. A Theory of Timed Automata. In *Theoretical Computer Science*, volume 126, pages 183–235. Elsevier Science (1994).

[11] Pearl, J. Heuristics: Intelligent Search Strategies for Computer Problem Solving. pages I–XVII, 1–382. Addison-Wesley Longman Publishing Co., Inc. (1984).

[12] Hart, P. E., Nilsson, N. J., and Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In *IEEE Transactions on Systems, Science and Cybernetics*, volume SSC-4, pages 100–107 (1968).

[13] Wehrle, M., Kupferschmid, S., and Podelski, A. Useless Actions are Useful. In *ICAPS*, pages 388–395 (2008).

[14] Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. Model Checking and the State Explosion Problem. In *LASER: LNCS 7682*, pages 1–30. Springer-Verlag (2012).

[15] Wehrle, M. *Transition-Based Directed Model Checking*. Ph.D. thesis (2011).

[16] Wehrle, M. and Kupferschmid, S. Mcta Directed Model Checking for Real-time Systems (2016). URL http://gki.informatik.uni-freiburg.de/tools/mcta/. Accessed: 09.01.2017.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Kevin Grimm

Matriculation number — Matrikelnummer
12-058-053

Title of work — Titel der Arbeit
Under-Approximation Refinement for Timed Automata

Type of work — Typ der Arbeit
Bachelor's thesis

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 12.01.2017

[illegible]

Signature — Unterschrift