Finding Optimal Plans for Domains with Restricted Continuous Effects with UPPAAL CORA

Henning Dierks
University of Oldenburg
Germany

Abstract

We present a translation of a variant of PDDL with restricted continuous effects into linearly priced timed automata. For the latter notion the model-checker UPPAAL CORA is able to find cost-optimal traces. We explain the PDDL variant and its translation into the syntax of UPPAAL CORA. A case study is used to explain the approach.

Introduction

An interesting trend in recent years in computer science research is the growing exchange of techniques in two areas which were rather disjoint previously. On the one side the planning community dealt with the problem to find valid plans in domains automatically. A main obstacle is to find informative and efficient heuristics to guide the search towards the goal. Usually the aspects of quantitative time and optimality of plans were neglected because finding just a valid plan in acceptable time was difficult enough in domains with huge state spaces.

On the other side the verification community worked hard to cope with the problem of state explosion when model-checking is applied. The standard technique is to find efficient data structures for symbolic representation of set of states. In case of discrete time this is rather successful and even for models with continuous time excellent tools are available, for example UPPAAL(?, ?). The common problem of both research areas are huge state spaces. Planning usually means to find a way through this space. Verification usually means to prove the absence of such a way. On the first sight it seems that the problems are contrary but there are situations where model-checking can benefit from heuristics. For example, when the model-checker

• tests a system that is known to be faulty or
• should check whether a system is able to execute a given abstract trace. This is rather often a problem when large systems are abstracted due to limited resources. When the model-checker finds an abstract trace the question is still open whether the abstraction was too coarse or the trace is feasible in the full model.

In both cases it make sense to guide the search of the model-checker. Examples of such approaches are (?; ?).

However, the planning community can benefit from the achievements of the verification community as soon as quantitative time and the question for optimality comes into the play. This is topic of this paper. We introduce a variant of the standard specification language for planning problems PDDL. It is an extension of PDDL 2.1 at level 3 towards duration-dependent and continuous effects. The latter effects, however, are restricted to the costs. That means it is allowed to specify the costs of a durative action depending of the duration of the action. To solve the problem of finding cost-optimal plans in such domains we translate the planning problem to linearly priced timed automata for which the tool UPPAAL CORA exists. It is able to find cost-optimal traces which represent valid plans.

The paper is organised as follows. You are reading the introduction. The next section introduces briefly our variant of PDDL. Thereafter we explain (priced) timed automata and the capabilities of the model-checker UPPAAL. The translation from PDDL to timed automata is explained in the following section. We end with a case study and conclusions.

PDDL

PDDL (Planning Domain Definition Language) has been introduced in (?) as common problem-specification language for the AIPS-98 planning competition. The purpose of PDDL is to describe the nature of a domain by specifying its entities and actions that may have effects on the domain. These effects can change the state of the domain.

In order to handle domains with time and numbers PDDL has been extended hierarchically (?; ?). The original PDDL is called PDDL level 1; the first extension by numeric effects represents level 2. That means in PDDL at level 2 it is possible to handle functional expressions and effects may change the values of those. The next extension (level 3) allows to specify durations for actions. Further extensions introduce duration-dependent effects for durative actions (level 4) and continuous effects for durative actions (level 5).

In this paper we will deal with PDDL at level 3 together with restricted continuous effects for durative actions (called PDDLcora) for which we offer an automatic translation from PDDLcora domain specifications into networks of timed automata suited for the real-time verification tool UPPAAL.
CORA. Thus, the entire collection of data structures and heuristic search-algorithms developed within the framework of UPFAAL become available to any planning problem describable within PDDLcora. An interesting aspect in which our approach differs from (?) is the fact that we leave the decidable set of timed automata due to our extensions. The reason is that by duration-dependent effects we can sum up durations. Thus, we are able to build “stopwatches” and it is known that timed automata with stopwatches are more expressive than timed automata (?). Moreover, reachability is undecidable for stopwatch automata.

In Fig. 1 an example of a domain description in PDDLcora is given. It describes a variation of a planning problem in which landing planes have to be scheduled to runways. Each plane has three landing times given:

- **earliest** denotes the earliest time where the aircraft can land.
- **target** describes the desired time. When the aircraft misses this time it is late.
- **latest** defines the latest time at which the plane must land.

Moreover each aircraft is assigned two rates. The optimal time point for the plane is the target time. In this case it produces no costs at all. Landing earlier than the target time increases the costs by the early-rate. Landing after the target time produces immediate costs, a so-called late-penalty. The costs for landing late increase with the late-rate.

In PDDLcora we model this problem as follows. We introduce two types plane and runway. Each plane should be landed in the goal state and landing requires that a runway is not occupied and the plane is scheduled to this runway. The procedure to land an aircraft requires to schedule the aircraft to a runway first. This is done by the schedule action and it action takes 40 time units. After landing the aircraft has to leave the runway which takes 40 time units also.

The landing is modelled by two actions. The first durative action is land-on-time which covers the case that plane lands in-between the earliest and target time. It starts at the earliest landing time and finishes at the target at latest. This is achieved by specifying

\[
(at \ start \ (= \ total-time \ (earliest \ ?p)))
\]

and

\[
(:\duration
\ (\leq \ ?duration \ (- \ (target \ ?p)
\ (earliest \ ?p))))
\]

The costs depend on the time when this action ends. We compute them by increasing the costs at the beginning and decrease them by the early rate during execution.

The second action models a delayed landing of the aircraft. In this case the action starts at the target time and takes at most the difference between latest and target time. The penalty for the delayed landing is paid at the beginning of the action and during the action the costs increase with the late-rate.

Figure 4 contains a simple system of three automata – called processes in UPFAAL – $P, Q$ and $R$ which work in parallel. The initial states $p_1, q_1$ and $r_1$ are marked by ingoing edges with no source. Initial all three clocks $(x, y, z)$ are set to 0 and no transition is enabled: The transition from $p_1$ to $p_2$ is blocked by the guard $x > 4$. The transition from $q_1$ to $q_2$ is also blocked by the guard on clock $y$. Similarly,

\[500 + (178 - 155) \cdot 10 = 530\]

as costs for this example plan.

**UPFAAL CORA**

UPFAAL$^1$ is a modeling, simulation, and verification tool for real-time systems modeled as networks of timed automata (?, ?) extended with data types such as bounded integer variables, arrays etc. In this subsection we describe briefly how networks of timed automata are specified in the syntax of UPFAAL. For more details about this tool the reader is confered to (?,?).

Figure 4 contains a simple system of three automata – called processes in UPFAAL – $P, Q$ and $R$ which work in parallel. The initial states $p_1, q_1$ and $r_1$ are marked by ingoing edges with no source. Initial all three clocks $(x, y, z)$ are set to 0 and no transition is enabled: The transition from $p_1$ to $p_2$ is blocked by the guard $x > 4$. The transition from $q_1$ to $q_2$ is also blocked by the guard on clock $y$. Similarly,

$^1$See the web site http://www.uppaal.com/.
(define (domain Planes)
  (:types plane runway)
  (:predicates (occupied ?r - runway)
    (landed ?p - plane)
    (scheduled ?p - plane ?r - runway))
  (:functions (earliest ?p - plane)
    (target ?p - plane)
    (latest ?p - plane)
    (early-rate ?p - plane)
    (late-rate ?p - plane)
    (late-penalty ?p - plane)
    (cost))

  (:durative-action schedule :parameters (?p - plane ?r - runway)
    :duration (= ?duration 40)
    :condition (at start (and (not (occupied ?r))
                               (not (landed ?p))
                               (not (scheduled ?p ?r))))
    :effect (and (at start (occupied ?r))
                 (at end (scheduled ?p ?r))))

  (:durative-action clear :parameters (?p - plane ?r - runway)
    :duration (= ?duration 40)
    :condition (and (at start (occupied ?r))
                    (at end (occupied ?r))
                    (at start (scheduled ?p ?r))
                    (at start (landed ?p)))
    :effect (and (at end (not (occupied ?r)))
                 (at start (not (scheduled ?p ?r))))

  (:durative-action land-on-time :parameters (?p - plane ?r - runway)
    :duration (<= ?duration (- (target ?p) (earliest ?p)))
    :condition (and (at end (occupied ?r))
                    (at end (scheduled ?p ?r))
                    (at start (= total-time (earliest ?p))))
    :effect (and (at end (landed ?p))
                 (at start (increase cost (* (early-rate ?p)
                                          (- (target ?p) (earliest ?p))))
                  (decrease cost (early-rate ?p))))

  (:durative-action land-late :parameters (?p - plane ?r - runway)
    :duration (<= ?duration (- (latest ?p) (earliest ?p)))
    :condition (and (at end (occupied ?r))
                    (at end (scheduled ?p ?r))
                    (at start (= total-time (target ?p))))
    :effect (and (at end (landed ?p))
                 (at start (increase cost (late-penalty ?p)))
                 (increase cost (late-rate ?p))))
)

Figure 1: The planes domain.
the transition from $r_1$ to $r_2$ is initially blocked because it requires a synchronisation on a channel $b$. Hence, the only legal behaviour of this system in the initial state is to let time pass. All clocks are incremented linearly with rate 1.

As soon as more than 4 time units have passed, the transition to $p_2$ can be fired. If this happens the clock $x$ will be reset to 0. Note that the system is not forced to execute this transition immediately after it is enabled. It may even wait forever unless an invariant at the source state is given. In this case $p_1$ is equipped with the invariant $x \leq 11$ and therefore the process $P$ is allowed to select a time point in $[4, 11]$ to fire the transition.

In state $p_2$ process $P$ can fire the transition to $p_3$ only together with another transition since it requires a synchronisation on channel $a$. The synchronisation on a channel $c$ is possible if there are two processes with transitions labelled with $c$ resp. $c'$ and both guard are satisfied. Then both transitions are executed together whereas the actions of the $c$ transition are executed before those of the $c'$ transition. Nevertheless the results of the actions are computed without any delay. In the example of Fig. 4 the system may switch from $p_2$ and $q_1$ to the states $p_3$ and $q_2$ simultaneously provided that $y == 13$ and $x == 6$ hold. When executed the variables $k$ and $m$ are both set to 42. It is easy to see that this can only happen when $P$ has taken the transition to $p_2$ at the time point 7.

If $Q$ has reached $q_2$ it has to leave this state at the same point in time because $q_2$ is marked as committed state. That requires the process to leave this state before time passes again. Although this behaviour can be expressed by setting the clock $y$ to 0 and adding the invariant $y \leq 0$, this feature is useful since in can save clocks which is the most expensive entity when model-checking timed automata. Hence, $Q$ is forced to synchronise on channel $b$ and process $R$ can serve as synchronisation partner. Hence at time point 13 but after the transition to $q_2$ it fires the transition to $q_3$ together with $R$ which enters $r_2$. The action sets $n$ to 42, too. Finally, $R$ may change from state $r_2$ to $r_3$ provided that $z > 28$ holds.

This example highlights only some features of the extended timed automata which can specified in UPPAAL. Further additions are extensions of the data types (enumerations, bounded integers and Boolean variables and arrays of those). In the latest versions also broadcasting channels are allowed. If a broadcast $c$! on channel $c$ is executed all processes which have a enabled, $c'$-labelled transition will take part.

In Fig. 5 the model of Fig. 4 is given in the ASCII syntax of UPPAAL. Queries are given in a different file and the tool is basically able to check for reachability properties.\footnote{Some extension in the expressiveness of the queries have been made but they are not used in our approach.} An example suitable for Fig. 4 is:

\[
\Sigma<> (P.p3 \text{ and } Q.q3 \text{ and } R.r3) \backslash A[m==k]
\]
Both queries are satisfied. Note that the last query states that \( m \) and \( k \) are always equal. Hence, the assignments of the \( a \)-synchronised transitions is executed atomically.

In more recent work (\(^1\), \(^2\)) the timed automata model as well as the underlying verification engine of UPPAAL have been extended to support computation of optimal reachability with respect to various cost criteria. The name of this tool variant is called UPPAAL CORA. In (\(^2\)) the timed automata model has been extended with discrete costs on edges and the optimality criteria consist in minimising either the total accumulated time (for reaching a goal state) or the total accumulated discrete cost or the sum of these two. UPPAAL CORA (\(^1\), \(^2\)) offer various mechanisms for guiding and pruning the search for optimal reachability and has been applied successfully on a number of scheduling problems (e.g. job-shop scheduling, air-craft landing).

An example of a priced timed automaton is given in Fig. 6. It models a typical scheduling problem of a traveller from Paris to London. There are two choices for her: Flying directly to optimise time consumption only. It takes 20 time units to fly directly and 30 time units via Amsterdam. If time is costly, i.e. one time unit costs one cost unit, then it is guaranteed that the first successful trace is also the cheapest one\(^3\).

A further extension of priced timed automata is presented in (\(^3\)). It is possible to define the costrate depending on the current system state which consists of a vector of states of all components. Hence, the current costrate is a linear sum of costrates. Therefore, this extension is called linearly priced timed automata (LPTA).

### Translation

In this section we explain how to translate the PDDL specifications into input for UPPAAL CORA. In contrast to (\(^1\)) we can exploit additional expressive power in the syntax of the target and gain therefore readability. Instead of a formal approximation.

\(^1\)We assume that \( remaining \) represents an under-
Figure 6: An example of a priced timed automaton.

clock x,y,z;
int k,m,n;
chan a,b;

process P {
    state p1 { x <= 11 },
        p2,p3;
    init p1;
    trans
        p1 -> p2 { guard x>4;
                    assign x:=0; },
        p2 -> p3 { guard x==6;
                    sync a?;
                    assign m:=k; };}
process Q {
    state q1,q2,q3;
    commit q2;
    init q1;
    trans
        q1 -> q2 { guard y==13;
                    sync a!;
                    assign k:=42;},
        q2 -> q3 { sync b!; };}
process R {
    state r1,r2,r3;
    init r1;
    trans
        r1 -> r2 { sync b?;
                    assign n:=k; },
        r2 -> r3 { guard z>28; };}

Figure 5: The example of Fig. 4 in the syntax of UPPAAL.

treatment we explain the translation by the example of the Landing Domain.

Global Aspects

For the predicates and functions of the domain we add appropriate (global) variables with appropriate type. Since the type system of UPPAAL CORA supports arrays we can simply produce the following declarations of global variables for the Landing domain.

bool occupied[ALL_OF_runway];
bool landed[ALL_OF_plane];
bool scheduled[ALL_OF_plane][ALL_OF_runway];
meta int earliest[ALL_OF_plane];
meta int target[ALL_OF_plane];
meta int latest[ALL_OF_plane];
meta int early_rate[ALL_OF_plane];
meta int late_rate[ALL_OF_plane];
meta int late_penalty[ALL_OF_plane];

It is clear that predicates are translated to Boolean and functions to integers. The sizes of the arrays depend on the type of the parameters. For example, for occupied we need as many Boolean variables as runways are defined in the problem. ALL_OF_runway is a constant that is declared before and depends on the problem specification. In our example we get the following constants:

// type plane
const int ALL_OF_plane = 3;
// type runway
const int ALL_OF_runway = 1;

An interesting feature of UPPAAL CORA is the definition of meta variables. Whenever the model-checker computes a new state it compare this new state with all previously seen states. However, some variables are considered to be auxiliary only. Hence, it can make sense to leave these variables out when checking a new state. This feature speeds up checks and reduces the stored state space since the tool does not distinguish states which differ only in these auxiliary variables. By the prefix meta we can specify such variables. In our translation we exploit this in the case of predicates and function in the domain which are never changed. These variables are only set once, namely by the problem
specification. Because all functions of the Landing domain are never changed by the domain’s actions we can declare them as meta.

**Durative Actions**

For the translation of durative actions we use process templates of UPPAAL CORA. That means that each durative action is translated into a corresponding template and in the final system declaration in UPPAAL CORA we instantiate some of these templates appropriately. Note that this differs completely from the way the translation was done in (?) because here we get a timed automaton for each instance of a template while in (?) integer variables where used. The advantage of this new approach is the direct correspondence between the PDDL specification and its translation into a template.

These templates can have parameters and the only parameter in our case is an unique identifier. We will explain our translation by the land_late action. We get the following process declaration:

```plaintext
process land_late(const int id) {
  int p, r;
  clock duration;
  int costrate=0;
  int min_duration, max_duration;

  The parameters of the durative action \(?p\) and \(?r\) occur here as local integer variables \(p\) and \(r\). In order to measure the duration of the action we add a local clock and two
variables \(\text{min\,duration}\) and \(\text{max\,duration}\) which are needed to record those duration constraints which are given at the begin of the durative action. We also add a variable
\(\text{costrate}\) that defines the rate in which the costs are increased (or decreased resp.) while executing the action.

**State Space:** Next is the state space declaration of the template. The basic idea is that the process starts in a state \(\text{idle}\) where the action is not executed. In order to start the durative action the process guesses instances of the action’s parameters \(?p\) and \(?r\). This is done by transitions through two auxiliary states called \(\text{guess1}\) and \(\text{guess2}\). The intention is that these states are left at same time point as they are entered. Finally, a state \(\text{work}\) is reached that means the durative actions is executed. To end the action a transition to \(\text{idle}\) is fired. In sum, we have the following state space declaration:

```plaintext
state idle,
  guess1 { duration <=0},
  guess2 { duration <=0},
  work { duration<=max_duration
          & & cost’==costrate };

commit guess1, guess2;
init idle;
```

This specifies that the automaton can stay in state \(\text{idle}\) arbitrarily long whereas the guess-states have to be left within 0 time units. The time it may stay in \(\text{work}\) is restricted by \(\text{max\,duration}\) only, since here only upper bounds are legal in UPPAAL CORA. The lower bounds are implemented by transition guards. The cost rate is also specified here. It simply states that the cost rate is given by the variable \(\text{costrate}\). Hence, we only have to manipulate this variable to change the current cost rate. In order to minimise the search space we also declare the \(\text{guess\,-states}\) as committed. Roughly, the meaning is that the system of all processes first fires transitions first which leave committed locations. That reduces the possible interleavings and saves search space. The declaration of \(\text{idle}\) as initial state is clear.

**Action Parameters:** Actions in PDDL may have parameters and our template for UPPAAL CORA implements this in the following way. Each action parameter is represented as local variable and for each parameter \(i\) we have an auxiliary state \(\text{guess}_i\) that guesses the current instance of the parameter \(i\), i.e. for each legal value \(j\) of the \(i\)th parameter we get an unrestricted transition from \(\text{guess}(i-1)\) to \(\text{guess}\) where the \(i\)th parameter is set to \(j\). For \(i = 1\) we identify \(\text{idle}\) with \(\text{guess0}\).

**Duration Constraints:** In PDDL the duration of durative actions are constrained by inequalities restricting the special variable \(?duration\). It is possible to use functions in these inequalities and hence the evaluation may change during execution of the action due to interference by other actions. Therefore, duration constraints are either evaluated at end or at start where the latter is the default. To models this in UPPAAL CORA we have to store the most restrictive bounds in the local variables \(\text{min\,duration}\) and \(\text{max\,duration}\). The check whether all duration constraints are satisfied can only happen when the state \(\text{work}\) is left. Hence, we get the following transitions. Note that only the parts of the transitions are shown which are relevant for the implementation of the duration constraints:

```plaintext
trans
  idle -> guess1 {
    guard ...
    assign ... duration:=0, ...
  },
  guess1 -> guess2 {
    ...
  },
  ...
  guess2 -> work {
    guard ...
    assign min_duration:=0,
    max_duration:=MAX\_DURATION,
    max_duration:=
    min(max_duration,
        (latest[p]-earliest[p])),
    ...
    duration:=0;},
  work -> idle {
    guard duration>=min_duration
    & & duration<=max_duration;
    assign ...
```

min_duration:=0,
max_duration:=0,
...

In our example action we only have an upper bound which has to be evaluated at start. Therefore it is computed in the transition from guess2 to work. The preceding assignments min_duration:=0,
max_duration:=MAX_DURATION make sure that both variables are initialised properly. MAX_DURATION is the maximal positive integer constant. When work is entered duration is reset to measure the time the action is executed currently. To stop this execution all constraints have to be satisfied. Hence, the guard of the transition from work to idle contains the check whether the duration is between min_duration and max_duration. This implements the at start constraints. The land-late actions has no at end duration constraints. But if it had such constraints they would appear here because they have to be evaluated exactly in the moment when the transition takes place.

Conditions and Non-Continuous Effects: Similar to duration constraints both conditions and effect come with a time specification in case of durative actions. The translation of conditions with at start and at end specification is straightforward. The syntax of guards for transition in UPPAAL CORA has been extended recently to simple C syntax. Hence it is basically an exchange of syntax because all PDDL operators have a representative in UPPAAL CORA syntax. The at start-conditions are checked when the transition to state work is executed. The at end-conditions are checked in the guard of the work-to-idle transition.4

An extension to standard PDDL is the possibility of adding total-time conditions in durative actions. The syntax is

(at start (comp total-time f_expr)
(at end (comp total-time f_expr)

where comp is a comparison operator (=,≥,≤,<,>) and f_expr is a functional expression. The obvious meaning is that the total-time of the system and the value of the functional expression should be in the comparison relation. Both the check and the evaluation of the functional expression take place as specified at start or at end respectively. The translation is straightforward by appropriate comparison with a global clock measuring the total time.

In the case of non-continuous effects the translation has to cope with the simultaneous execution of all effects. In case of Boolean variables this is no problem since it is clear to which value a Boolean variable is set by an effect. However, in case of functions our translation has to translate

(and (assign a b) (assign b a))

into a sequence of non-simultaneous assignments that switch the values of the nullary functions a and b. A naive translation using auxiliary variables would add an unnecessary blow up of the search. However, the new feature of meta variables in UPPAAL CORA allows us to add for each function f (in PDDL) not only the integer variable f but also a meta variable new_f without any price to pay. That means that the example above would be translated to

new_a:=b, new_b:=a, a:=new_a, b:=new_b

in UPPAAL CORA. A special treatment is necessary for the variable cost. It has a special meaning in UPPAAL CORA and it must not occur in functional expressions. In our PDDL variant non-continuous assignments to cost are legal but must use the increase operator. The specification of the penalty that a late plane has to pay is

(at start (increase cost (late-penalty ?p)))

and it is translated into the assignment

cost+=late_penalty[p]

which is placed at the assignment of the transition to state work.

Continuous Effects for Costs: Our translator deals with a restricted set of continuous effects. It is possible to specify cost rates for a durative action in the following way

(increase cost f_expr)
(decrease cost f_expr)

Here, f_expr stands for an arbitrary functional expression. The meaning is that the costs for the durative action depend on the duration actually needed and (positive or negative resp.) rate is given by f_expr. Note that more than one durative action may be active at the same time. But we can exploit that UPPAAL CORA is able to cope with cost rates for each process. The translation of an effect (increase cost f_expr) is just an assignment to the local variable costrate. In our example we get

costrate=late_rate[p]

which has to be placed at the assignment of the transition to state work.

The Problem Definition

Above we discussed the aspects of the domain translation but the current problem is given by a problem specification. The contents of the problem specification can be translated in a very canonical way. For the problem we add a process called problem with three states: initial, work, goal. The transition from initial to work initialises all predicates and functions as specified in the (:init ...) part of the problem definition. The transition from work to goal can only be fired if the property given in the (:goal
Optimisations

The translation introduces further variables in order to avoid searching invalid plans. This happens when the guessing part of a process for a durative action selects instances of the parameters which do not satisfy the conditions given in the domain specification. In this case the only transition left for the process is a transition to idle where a special global Boolean flag \texttt{blocked} is set. As soon as this flag is set all actions cannot leave the \texttt{idle} state anymore. The effect is that the computation stops and \textsc{Uppaal} \textsc{Cora} will not invest any time anymore in this branch.

Another important aspect is the number of process instances per template. It is clear that durative actions may overlap and the model-checker should consider such plans since they are often less time consuming than subsequent plans. Therefore it is reasonable to have several instances per template. However, increasing the number of processes increases the search space significantly. To minimise the price we have to pay for additional instances we added the following construction. If a durative action \texttt{da} is represented \( n \) times in the system for \textsc{Uppaal} \textsc{Cora}, then we have an Boolean array of size \( n \) representing the information whether an instance is currently executing its action. When model-checker wants to start a new action it has to select the unique instance that is currently not active and has the minimal \texttt{id} of such inactive instances. This avoids that the tool considers plans which are equivalent up to matching to process instances.

Case Study

We made a series of experiments based on the planes domain.

<table>
<thead>
<tr>
<th># Planes</th>
<th># instances</th>
<th># clocks</th>
<th>plan</th>
<th>time</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>1.5</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>13</td>
<td>0</td>
<td>28</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>9</td>
<td>860</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>13</td>
<td>860</td>
<td>249</td>
<td>479</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>9</td>
<td>1540</td>
<td>36</td>
<td>122</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>13</td>
<td>out of mem (165 s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>9</td>
<td>out of mem (123 s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>13</td>
<td>180</td>
<td>1051</td>
<td>788</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>out of mem (139 s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>out of mem (139 s)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We translated each problem specification (the number of planes is given in the first column of the table above) in two ways. The first variant contained two instances of each action template, the second variant had three instances. Consequently we get either 9 resp. 13 clocks in the input of \textsc{Uppaal Cora}. Since the number of clocks is the most expensive entity w.r.t. complexity we only added information in the table. The result of the model-checking is given in the last three columns. In case of success they contain the \textit{cost} of the optimal plan, the cpu time needed (on a Xeon with 2.66 GHz with memory limit set to 900MB) in seconds and the memory needed in MB.

It is an important aspect that \textsc{Uppaal Cora} was not guided at all since our prototype does not construct any additional informative heuristics, i.e. \textit{remaining} and \textit{heur} are not used. Hence, only pruning takes place as soon as a plan was found. It is obvious that appropriate planning techniques can be applied to improve these results. First attempts in this direction are made in the AVACS project\footnote{see \url{http://www.avacs.org} for more information about this project}.

All files of experiments are available at the website of the author:

http://csd.informatik.uni-oldenburg.de/dierks/

Conclusion

It was shown that due to the recent improvements of \textsc{Uppaal Cora} it makes sense to consider this tool as planning tool for optimal planning problems provided that the continuous effects are restricted to costs only. A matter of future work is to find out to what extend this approach can be extended to less restricted continuous effects.

A key factor whether our approach should be considered for a certain domain is the relation between discrete state space and continuous state space. In many domains – for example those of IPC 2004 – the discrete state space is predominant even in domains where durative actions are used. From our perspective this is mainly due to the evolution of the planning community from the search in huge discrete state spaces. In case of purely discrete domains the planning tools are very powerful since much effort has been spent in the past on such domains. In contrast to that a real-time model-checker like \textsc{Uppaal} has been tailored to deal with continuous state spaces and it is still an open research issue to find efficient symbolic representations of mixed states spaces. Hence, \textsc{Uppaal Cora} is a good option for domains where the continuous state space is predominant. As soon as durations and continuous effects are missing or are a minor part of the domain our approach will not be competitive.

Acknowledgements:

The author thanks Gerd Behrmann, Kim G. Larsen and Jacob Illum Rasmussen from Aalborg for supporting this work by building \textsc{Uppaal Cora}, providing case studies and several suggestions.

Moreover he thanks R. Howey, D. Long and M. Fox from Strathclyde for distributing the plan validator VAL as open
source. The translator the author built is based on the parser for PDDL which is distributed together with VAL.