Planning as Model Checking in Hybrid Domains

Sergiy Bogomolov
University of Freiburg
Germany
bogom@cs.uni-freiburg.de

Daniele Magazzeni
King's College London
United Kingdom
daniele.magazzeni@kcl.ac.uk

Andreas Podelski
University of Freiburg
Germany
podelski@cs.uni-freiburg.de

Martin Wehrle
University of Basel
Switzerland
martin.wehrle@unibas.ch

Abstract

Planning in hybrid domains is an important and challenging task, and various planning algorithms have been proposed in the last years. From an abstract point of view, hybrid planning domains are based on hybrid automata, which have been studied intensively in the model checking community. In particular, powerful model checking algorithms and tools have emerged for this formalism. However, despite the quest for more scalable planning approaches, model checking algorithms have not been applied to planning in hybrid domains so far. In this paper, we make a first step in bridging the gap between these two worlds. We provide a formal translation scheme from PDDL+ to the standard formalism of hybrid automata, as a solid basis for using hybrid system model-checking tools for dealing with hybrid planning domains. As a case study, we use the SpaceEx model checker, showing how we can address PDDL+ domains that are out of the scope of state-of-the-art planners.

1 Introduction

Planning in hybrid domains is a challenging problem that has found increasing attention in the planning community. In addition to classical planning, hybrid domains allow for modeling continuous behavior with continuous variables that evolve over time. Such problems frequently occur in practice, e. g., in robotics or embedded systems. Furthermore, real-world scenarios must take into account that exogenous events may happen, as a consequence or independently of the plan actions. PDDL+ (Fox and Long 2006) is the PDDL extension for modeling such domains through the use of continuous processes and events.

Planning in hybrid domains is challenging because apart from the state explosion caused by discrete state variables, the continuous variables cause the reachability problem undecidable (Alur et al. 1995). From a practical point of view, various planning algorithms and tools with different features and limitations have emerged (Penberthy and Weld 1994; McDermott 2003; Li and Williams 2008; Coles et al. 2012; Shin and Davis 2005; Della Penna et al. 2009). However, only TM-LPSAT (Shin and Davis 2005) and UPMurphi (Della Penna et al. 2009) can deal with the full feature range of PDDL+, and both suffer from scalability issues.

From an abstract point of view, it is well known that hybrid planning domains are related to the formalism of hybrid automata (Henzinger 1996) studied in model checking. In the last years, powerful model checking techniques and tools based on, e. g., SMT (Cimatti et al. 2000) and symbolic search (Frehse 2008; Frehse et al. 2011), have been developed for this formalism. Apparently, algorithms based on such techniques can possibly be beneficial for planning in hybrid domains as well, and might particularly help to tackle the limitations of currently available planning systems with respect to the supported PDDL+ feature range. However, despite the relationship of hybrid planning domains and hybrid automata, these techniques have not been applied for planning in hybrid domains so far. The main obstruction to this synergy is the lack of a common modeling language, which makes it difficult to share benchmarks and to foster the cross-fertilization between these two areas.

In this paper, we make a first step in bridging the gap between these two worlds. We provide a formal translation from PDDL+ to the formalism of hybrid automata. The translation provides an over-approximation of the PDDL+ semantics, which is sufficient to prove *plan non-existence* in unsolvable domains. In addition, we identify a subset of PDDL+ features for which our translation is exact and can be applied for finding hybrid plans. In contrast to the class of hybrid automata that has been used to define the semantics of PDDL+ (Fox and Long 2006), our translation obeys the *standard* semantics of hybrid automata. A case study with the SpaceEx model checker (Frehse et al. 2011) shows considerable improvements in scalability compared to the Colin and UPMurphi planner, and extends the class of tractable problems. Overall, our translation is supposed to build a solid basis for using hybrid system model checking tools for dealing with hybrid domains, thus extending the planning-as-model-checking paradigm (Cimatti, Roveri, and Traverso 1998) to the domain of hybrid systems.

2 Preliminaries

In this section, we provide the background and a description of the formal models we use throughout the paper.

2.1 The PDDL+ Language

PDDL+ supports the representation of domains with a mixed discrete-continuous dynamics, providing a flexible model of continuous change. In particular, it allows to model exogenous events to reflect changes that are initiated by the environment. PDDL+ is built on top of PDDL 2.1 and introduces the new constructs of *processes* and *events*.

Definition 1 (Planning Instance). *A planning instance is a pair* $I = (Dom, Prob)$*, where* $Dom = (Fs, Rs, As, Es, Ps, arity)$ *is a tuple consisting of a finite set of* function symbols Fs*, a finite set of* relation symbols Rs*, a finite set of (durative)* actions As*, a finite set of* events Es*, a finite set of* processes Ps*, and a function arity mapping all symbols in* $Fs \cup Rs$ *to their respective arities.*

The triple $Prob = (Os, Init, G)$ *consists of a finite set of* domain objects Os*, the* initial *state Init, and the* goal *specification* G*.*

For a given planning instance Π, a *state* of Π consists of a discrete component, described as a set of propositions P, and a numerical component, described as a vector of real variables **v**. Instantaneous actions are described through preconditions (which are conjunctions of propositions in P and/or numeric constraints over **v**, and define when an action can be applied) and effects (which define how the action modifies the current state). *Instantaneous* actions and events are restricted to the expression of discrete change. Events have preconditions as for actions, but they are used to model exogenous change in the world, therefore they are triggered as soon as the preconditions are true. A process is responsible for the continuous change of variables, and is active as long as its preconditions are true.

Durative actions have three sets of preconditions, representing the conditions that must hold when it starts (denoted by $\text{pre}_{\vdash}$ and $\text{pre}_{\vdash}^{num}$ to distinguish between preconditions on propositions and on numeric constraints, respectively), the invariant that must hold throughout its execution (propositional invariant $\text{pre}_{\leftrightarrow}$ and numeric invariant $\text{pre}_{\leftrightarrow}^{num}$), and the conditions that must hold at the end of the action ($\text{pre}_{\dashv}$ and $\text{pre}_{\dashv}^{num}$). Similarly, a durative action has three sets of effects: effects that are applied when the action starts ($\text{eff}_{\vdash}^{+}$, $\text{eff}_{\vdash}^{-}$, $\text{eff}_{\vdash}^{num}$ denoting predicates that are added, deleted, and numeric effects, respectively), effects that are applied when the action ends ($\text{eff}_{\dashv}^{+}$, $\text{eff}_{\dashv}^{-}$, $\text{eff}_{\dashv}^{num}$) and a set of continuous numeric effects $\text{eff}_{\leftrightarrow}^{num}$ which are applied continuously while the action is executing. A graphical representation of a durative action is shown in Fig. 1.

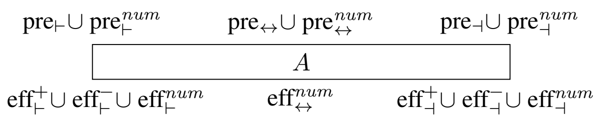

$\text{pre}_{\vdash} \cup \text{pre}_{\vdash}^{num}$
$\text{pre}_{\leftrightarrow} \cup \text{pre}_{\leftrightarrow}^{num}$
$\text{pre}_{\dashv} \cup \text{pre}_{\dashv}^{num}$
A
$\text{eff}_{\vdash}^{+} \cup \text{eff}_{\vdash}^{-} \cup \text{eff}_{\vdash}^{num}$
$\text{eff}_{\leftrightarrow}^{num}$
$\text{eff}_{\dashv}^{+} \cup \text{eff}_{\dashv}^{-} \cup \text{eff}_{\dashv}^{num}$

Figure 1: PDDL durative action

A durative action A has a duration $dur(A)$ which can either be fixed in the model or left to the planner decision.

2.2 Hybrid Automata

Hybrid automata (Henzinger 1996) are subject to active research. Intuitively, hybrid automata are finite state automata extended with continuous variables that evolve over time.

Definition 2 (Hybrid Automaton). *A* hybrid automaton *is a tuple* $\mathcal{H} = (Loc, Var, Init, Flow, Trans, I)$*, where Loc is a finite set of locations,* $Var = \{x_1, \ldots, x_n\}$ *is a set of real-valued variables,* $Init(\ell) \subseteq \mathbb{R}^n$ *is the set of initial values for* $x_1, \ldots, x_n$ *for all locations* ℓ*. For each* $\ell \in Loc$*,* $Flow(\ell)$ *is a relation over the variables in* Var *and their derivatives*

$$\dot{x}(t) = Ax(t) + u(t), u(t) \in \mathcal{U},$$

where $x(t) \in \mathbb{R}^n$*,* A *is a real-valued* $n \times n$ *matrix and* $\mathcal{U} \subseteq \mathbb{R}^n$ *is a closed and bounded convex set. Trans is a set of discrete transitions* (ℓ, g, ξ, ℓ')*, where* ℓ *and* ℓ' *are the source and target locations,* g *is the guard of* t *(given as a linear constraint), and* ξ *is the update of* t *(given as an affine mapping).* $I(\ell) \subseteq \mathbb{R}^n$ *is an invariant for all locations* ℓ*.*

The semantics of hybrid automata is defined as follows. A *state* of $\mathcal{H}$ is a tuple $(\ell, \mathbf{x})$ consisting of a location $\ell \in Loc$ and a point $\mathbf{x} \in \mathbb{R}^n$. More formally, $\mathbf{x}$ is a valuation of the continuous variables in Var. Let $T = [0, \Delta]$ be a time interval for some $\Delta \geq 0$. A *trajectory* of $\mathcal{H}$ from state $s = (\ell, \mathbf{x})$ to state $s' = (\ell', \mathbf{x}')$ is defined by a tuple $\rho = (L, \mathbf{X})$, where $L : T \rightarrow Loc$ and $\mathbf{X} : T \rightarrow \mathbb{R}^n$ are functions that define for each time point in T the location and values of the continuous variables, respectively. The trajectory ρ starts in $(\ell, \mathbf{x})$, ends in $(\ell', \mathbf{x}')$, and obeys the following constraints:

- The sequence of time points in ρ where the location is changed (according to L) increases strictly monotonically, starts with time point 0 and ends with time point Δ.
- There are no location changes not defined by L (i. e., locations are not changed during the continuous evolution).
- For all $t \in T$, the continuous variable evolution is consistent with the differential equation and invariant of $L(t)$.

A *network* $\mathcal{N} = \{\mathcal{H}_1, \ldots, \mathcal{H}_m\}$ of hybrid automata is a set of hybrid automata. The semantics of $\mathcal{N}$ is defined based on the semantics of single hybrid automata, with the following extensions. Every automaton in $\mathcal{N}$ is associated with a finite set of *synchronization labels*, including a special label τ in all label sets. The discrete component of a *state* s of $\mathcal{N}$ is defined as a *vector* of locations that denotes the current locations of every component in $\mathcal{N}$. Similarly, in addition to single automata, a *trajectory* of $\mathcal{N}$ maps time points to vectors of locations of each automaton. For a time point t, changes in the location vectors in a trajectory can either be caused by a single transition labeled with τ of one automaton in $\mathcal{N}$ ("interleaving transition"), or there are several automata in $\mathcal{N}$ that simultaneously fire transitions with equal synchronization labels $\neq \tau$ ("synchronized transition").

Informally speaking, global discrete update steps of $\mathcal{N}$ are either performed by single interleaving transitions of one automaton, or by firing several local transitions of several automata in a synchronized way. To perform such a synchronized transition via label c, it is required that *all* automata that have c in their synchronization alphabet *must* fire – if this is not possible, the global discrete update step is not possible either, i. e., the global transition is not applicable.

Although parts of PDDL+ are defined in terms of hybrid automata, there are several semantical issues raised by PDDL+ which do not allow to apply the translation given by Fox and Long (2006) to common model checking tools – in particular, their translation does not obey the standard semantics of hybrid automata. As a basis for our paper, we discuss these issues in more detail in the next section.

3 Semantical Issues Raised by PDDL+

Planning with continuous time is a challenging task, which becomes even harder when processes and events are present. In the formal definition of PDDL+ (Fox and Long 2006), assumptions about the class of domains that can be modeled and about plan validity (Fox, Howey, and Long 2005) are made. In the following, we briefly recall these assumptions.

No Moving Targets: The *no moving targets* rule states that no two actions are allowed to simultaneously make use of a value if one of the two is accessing the value to update it (i. e., the value is a moving target for the other action to access). As a consequence of this restriction, plans must respect the *ε-separation* requirement, i. e., interfering actions must be separated by at least a time interval of length ε. The planner Colin (Coles et al. 2012) makes a strictly stronger assumption, extending this requirement also to actions that are not mutex. In our work, we make the same assumption as Colin. We remark that *ε-separation* is not respected in the standard hybrid automata semantics, where transitions can start or end at the same instant and hence can compromise plan validity.

Events: Events are particularly challenging as they could trigger an infinite cascading sequence of events. To address this issue, we make the same restrictions proposed by Fox and Long (2006). Firstly, each event must delete one of its own preconditions and thus avoid self triggering. Secondly, planning instances must be *event-deterministic*: In every state in which two events e_1 and e_2 are applicable, the transition sequences e_1 followed by e_2 and e_2 followed by e_1 are both valid and reach the same resulting state.

Actions and events reveal the key difference between state changes that are deliberately planned (actions), and those that are caused by changes in the world (events). While the planner can decide whether or not to fire an applicable action (actions are *may* transitions), events have to be fired as soon as they become enabled (events are *must* transitions). This distinction complicates the relationship between PDDL+ and standard hybrid automata, where such a distinction is not present and all transitions are *may* transitions.

Concurrent Processes: It is possible that several processes are active at the same time, affecting the value of the same variable. To handle such *concurrent processes*, the continuous effects affecting the rate of change of a variable are combined by simply summing the effects of the processes. Although the handling of concurrent processes is very simple in PDDL+, it is a problematic feature in the standard hybrid automata setting, as each location in a hybrid automaton contains a single flow describing the continuous effects corresponding to that location. Therefore, combining the effects of concurrent processes would generate an explosion of the number of locations in the hybrid automaton.

PDDL+ Semantics: Fox and Long (2006) give a formal semantics of PDDL+ providing a mapping between PDDL+ domain and hybrid automata. However, Fox and Long make the key assumption to have hybrid automata where *conditional flows* can be defined. In any given location, instead of having a fixed rate of change for each variable, the rate of change of each variable depends on which processes are active in the current state. Conditional flows allow for easily modeling concurrent processes by using different rates of change depending on the current state.

Similarly, conditional flows are used to model events. To model events, we must force the corresponding event automaton to leave the current location (and to enter the location corresponding to the event's effect) as soon as it is triggered. Hence, the issue is to model *must* transitions. For this purpose, Fox and Long use the *time slippage* mechanism. A time-slip variable $\mathtt{T}$ is used to measure the amount of time that elapses between the preconditions of an event becoming true and the event triggering. The value of $\mathtt{T}$ must be 0 in any valid planning instance. To this aim, each location contains an invariant enforcing this requirement. Furthermore, the conditional flow is extended with the additional time-slippage flow that sets $\dot{\mathtt{T}} = 1$ whenever the preconditions of any event become true. However, conditional flows are not part of standard hybrid automata semantics. Furthermore, the issue of modeling the *ε-separation* is not addressed.

4 Modeling PDDL+ as Hybrid Automata

Based on the discussed semantical issues raised by PDDL+, we provide a formal translation of hybrid planning domains to *standard* hybrid automata to overcome these limitations. For the description of our translation, we assume a grounded planning instance I and use the following naming conventions: Function symbols are denoted with *continuous variables*, whereas (Boolean) grounded predicates are denoted with *discrete variables*. In particular, for the rest of the paper, we assume the actions in I to be grounded. The translation from I to a network of hybrid automata is based on translating grounded actions, discrete and continuous variables, events and processes to corresponding hybrid automata. This translation is described in the next sections.

4.1 Discrete Variable Automata

Common model checkers like SpaceEx do not support discrete variables in their input models. Hence, we represent discrete variables with a variant of their domain-transition graphs (Helmert 2006). A Boolean variable v is translated to automaton $\mathcal{H}_v$ with two locations that reflect the *true* and *false* values of v. Transitions between locations reflect how values can be changed through actions. More precisely, the synchronization labels reflect the discrete preconditions and effects of actions that have v in their precondition and effect, respectively. Roughly speaking, labels c that do not occur in all transitions of $\mathcal{H}_v$ possibly require changing $\mathcal{H}_v$'s location in order to be able to synchronize with c (thus representing a precondition). Similarly, labels c that occur in at least one non-self loop transition of $\mathcal{H}_v$ reflect that c can possibly change the value of v (thus representing an effect). We will make the description of synchronization labels more precise when introducing the translation for actions.

4.2 Continuous Variable Automata

Continuous variables x are translated to automata $\mathcal{H}_x$ as follows. For all possible flows $\dot{x} = k$ of x, $\mathcal{H}_x$ contains a location annotated with $\dot{x} = k$. There is a transition between two locations if it is possible to change the flow of x accordingly via an action. Furthermore, for all actions that affect the particular flow, there is a self-loop in the corresponding location. As an example, consider the automaton in Fig. 2.

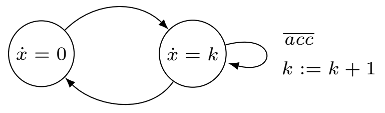
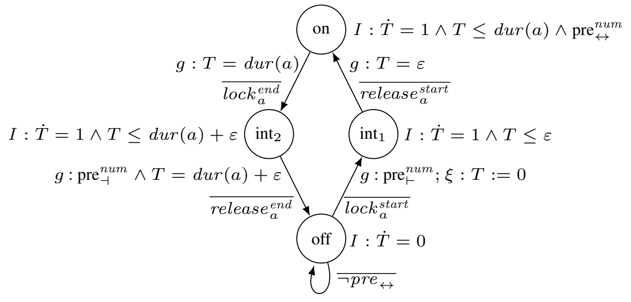


Figure 2: Example variable automaton $\mathcal{H}_x$

The automaton $\mathcal{H}_x$ models the behavior of the acceleration x of an engine. There are two possible flows for x, namely $\dot{x} = 0$ (corresponding to the case that the engine is turned off), and $\dot{x} = k$ (corresponding to the case where the engine is turned on, and the current acceleration is k). In case the engine is turned on, we can apply the action accelerate (represented as label $\overline{acc}$) to increase k.

4.3 Durative Action Automata

Grounded durative actions are translated to automata $\mathcal{H}_a$ such that $\mathcal{H}_a$ ensures the ε-separation property, and such that the (propositional and numeric) preconditions, effects and invariants of a are respected when a is starting, running and ending, respectively. For a given action a, $\mathcal{H}_a$ has the overall structure given in Fig. 3. The guards and updates of $\mathcal{H}_a$'s transitions are denoted with g and ξ according to Def. 2. The locations are annotated with the corresponding invariant I. Synchronization labels are annotated with a bar.



on
$I : \dot{T} = 1 \wedge T \leq dur(a) \wedge \text{pre}_{\leftrightarrow}^{num}$
$g : T = dur(a)$
$\overline{lock_a^{end}}$
$g : T = \varepsilon$
$\overline{release_a^{start}}$
$I : \dot{T} = 1 \wedge T \leq dur(a) + \varepsilon$
int$_2$
int$_1$
$I : \dot{T} = 1 \wedge T \leq \varepsilon$
$g : \text{pre}_{\dashv}^{num} \wedge T = dur(a) + \varepsilon$
$\overline{release_a^{end}}$
$g : \text{pre}_{\vdash}^{num}; \xi : T := 0$
$\overline{lock_a^{start}}$
off
$I : \dot{T} = 0$
$\overline{\neg pre_{\leftrightarrow}}$

Figure 3: Structure of action automaton $\mathcal{H}_a$

The automaton $\mathcal{H}_a$ uses a local continuous variable T that models a clock to keep track of action a's duration. Based on T, $\mathcal{H}_a$ simulates the execution of a as follows.

1. The *off* location models that a is not running. The invariant $\dot{T} = 0$ reflects that the clock T is stopped as well. (See below for a description of the self-loop.)
2. The int_1 location and the transition from *off* to int_1 model the behavior of a in the time interval $[0, \varepsilon]$ (for brevity, we assume that a is started at time point 0). The invariant of int_1 ensures that T is running, and that int_1 is left after at most ε time units. The guard g of the transition leading to int_1 reflects a's numeric precondition $\text{pre}_{\vdash}^{num}$, and its update ξ resets the clock T to zero. In addition, through synchronization, the label $\overline{lock_a^{start}}$ ensures the ε-separation property during the starting phase of a, as well as the required behavior of a's preconditions and effects:
   - In order to ensure the ε-separation property, $\overline{lock_a^{start}}$ locks the overall system in the sense that no other automaton can start (or end, see below) as long as $\mathcal{H}_a$ is in the int_1 location. To achieve this, a global lock automaton synchronizes with this label, with the property that such a synchronization is no longer possible for starting or ending other actions until the lock is released. To make this more clear, the lock automaton (simplified such that it only contains transitions for the starting phase of a) is depicted in Fig. 4 (see below for a description of the corresponding release label).

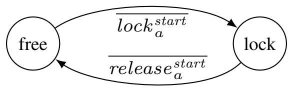


Figure 4: Global lock automaton

   - $\overline{lock_a^{start}}$ reflects the check for the propositional precondition $\text{pre}_{\vdash}$ as well as the check for the invariant $\text{pre}_{\leftrightarrow}$ (recall that $\text{pre}_{\leftrightarrow}$ must hold during the execution of a, hence it must hold at the start of a). These preconditions are satisfied iff a synchronization with corresponding discrete variable automata is possible. For example, if $\text{pre}_{\vdash}$ requires a variable v to be true, this is reflected in the corresponding variable automaton $\mathcal{H}_v$ as depicted on the left in Fig. 5, where we observe that v must be in the *true* location such that a synchronization is possible.

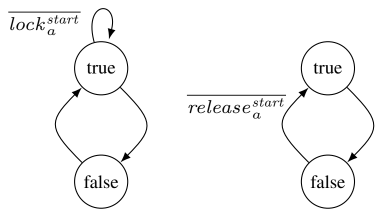


Figure 5: Example variable automata $\mathcal{H}_v$ and $\mathcal{H}_w$

   - $\overline{lock_a^{start}}$ reflects the continuous numerical effects described by $\text{eff}_{\leftrightarrow}^{num}$, which affects the flow of its continuous variables by synchronizing with the corresponding continuous variable automata (e. g., by increasing k in Fig. 2, where $\overline{acc}$ is replaced by the lock label).
3. The transition from int_1 to *on* models the time point ε. According to the guard $T = \varepsilon$, it must be taken after exactly ε time units. Furthermore, its label $\overline{release_a^{start}}$ releases the system via the lock automaton, allowing other

actions to start or end again. In addition, $\overline{release_a^{start}}$ reflects the start effects $\text{eff}_{\vdash}^{+}$, $\text{eff}_{\vdash}^{-}$ and $\text{eff}_{\vdash}^{num}$ by synchronizing with the corresponding variable automata. For example, if $\text{eff}_{\vdash}^{+}$ sets a variable w from false to true, this is reflected in $\mathcal{H}_w$ as shown on the right in Fig. 5.

4. The *on* location models a's behavior in the time interval $[\varepsilon, dur(a)]$. The invariant of *on* reflects the duration of a and the numeric invariant $\text{pre}_{\leftrightarrow}^{num}$. We remark that without further knowledge, doing so would require $\text{pre}_{\leftrightarrow}^{num}$ to hold in the time interval $[\varepsilon, dur(a)]$, whereas the original PDDL+ semantics would only require it to hold in the interval $[\varepsilon, dur(a) - \varepsilon]$. However, due to the ε-separation, the behavior of numeric invariants with strict and non-strict inequalities is identical, and we can hence interpret strict as non-strict inequalities without loss of generality.

5. Propositional invariants of actions a must hold as long as $\mathcal{H}_a$ is in its *on* location. To model this, we include all synchronization labels that possibly violate a's propositional invariant into the synchronization alphabet of $\mathcal{H}_a$ (e. g., for a propositional invariant $p = true$, we include all labels that represent effects of actions that set p to false). This causes all actions that execute an effect that violates a's propositional invariant to synchronize with a transition in $\mathcal{H}_a$. However, as there is no such outgoing synchronization transition of the *on* location, actions cannot violate a's propositional invariant as long as $\mathcal{H}_a$ is running. To be able to synchronize with such actions when a is not running, we introduce self-loops to $\mathcal{H}_a$'s *off* location that allow corresponding synchronization. In more detail, the self-loop with label $\overline{\neg pre_{\leftrightarrow}}$ in the *off* location represents a set of self-loops with labels for actions that violate a constraint in $\text{pre}_{\leftrightarrow}$. Note that we do not need such self-loops for int_1 (and int_2) because int_1 (and int_2) model the locked system where no other action may start or end.

6. The int_2 location and the transition from *on* to int_2 models the behavior of a in the interval $[dur(a), dur(a) + \varepsilon]$ when a is finished. The label $\overline{lock_a^{end}}$ locks the system to ensure the ε-separation during the ending phase of a, and reflects $\text{pre}_{\dashv}$ via synchronization (analogously to the start of a). In addition, it reflects the end of the continuous numeric change reflected by $\text{eff}_{\leftrightarrow}^{num}$. For example, if k has been increased by $\text{eff}_{\leftrightarrow}^{num}$ in Fig. 2 at the start of a, k is decreased again to reset the flow before a has been started (a corresponding self-loop is omitted in Fig. 2).

7. The transition from int_2 to *off* models the end of the execution of a. Its guard checks both a's duration $T = dur(a) + \varepsilon$ and precondition $\text{pre}_{\dashv}^{num}$. The label $\overline{release_a^{end}}$ releases the system (indicating that a is finished), and reflects the effect updates $\text{eff}_{\dashv}^{+}$, $\text{eff}_{\dashv}^{-}$, and $\text{eff}_{\dashv}^{num}$.

We observe that, by construction, the PDDL+ semantics of durative actions a is reflected by the hybrid automaton $\mathcal{H}_a$. In particular, $\mathcal{H}_a$ respects the ε-separation property.

4.4 Instantaneous Action Automata

Instantaneous actions a are modeled as automata $\mathcal{H}_a$ as follows. Similarly to durative actions, $\mathcal{H}_a$ contains an *off* and *on* location and respects the ε-separation. However, in contrast to durative actions, instantaneous actions do not feature durations (as suggested by the name), and hence, we do not need additional intermediate locations in the automaton model. In more detail, the transition from *off* to *on* is labeled with a corresponding $\overline{lock_a}$ label, which locks the system via the global lock automaton, and reflects the guard and effects analogously to the $\overline{lock_a^{start}}$ labels for durative actions (we do not need to distinguish between start and end labels because a is instantaneous). In addition, the transition features a numerical guard constraint that reflects the numerical precondition $pre_{\vdash}^{num}$ of a. Finally, $\mathcal{H}_a$ stays for ε time in *on*, and releases the system by returning to *off*.

4.5 Event and Process Automata

Events and processes require a *must* semantics, as they trigger as soon as they become enabled. In this paper, we over-approximate this *must* behavior with the (common) *may* behavior, which allows for more behavior and is hence sufficient for proving plan non-existence. Generally, over-approximations allow for at least the same (and possibly more) behavior as the original model. (Realizing *must* behavior more precisely is an important issue for future work).

Events are essentially instantaneous actions with *must* behavior. Hence, in our translation, we over-approximate events with instantaneous action automata.

Processes p are modeled as automata $\mathcal{H}_p$ that consist of an *off* and *on* location similar to events. There is a transition from *off* to *on* which synchronizes over the propositional precondition constraints $pre_{\vdash}$. This is an over-approximation because the transition is not forced to be taken as soon as possible. Furthermore, the *on* location features an invariant induced by the numeric precondition $pre_{\vdash}^{num}$. Finally, there are transitions from *on* to *off* for each negated constraint in $pre_{\vdash}^{num}$ (reflecting that the numeric invariant gets violated), and a transition that allows for returning in case the propositional invariant is set to false (again yielding an over-approximation). The effects of p are reflected in the same way as for continuous variable automata (e.g., see again Fig. 2). This translation allows modeling of concurrent processes in the standard hybrid automata semantics without the need of conditional flows.

4.6 Overall Translation Scheme

For a given planning instance, the overall translation is defined by a network of hybrid automata which contains a translated automaton for all discrete and continuous variables, durative and instantaneous actions, processes and events. The resulting system of hybrid automata is an over-approximation of the original PDDL+ planning instance.

Proposition 1. *Let I be a planning instance, and let $\mathcal{N}$ be the translated network of hybrid automata. Then for all plans π in I, there is a corresponding sequence σ of transitions in $\mathcal{N}$ such that for each time point t, the values of the discrete and continuous variables of π and σ are equal in t.*

Proof. (sketch) By construction, the semantics of variables and actions is reflected exactly by the translated automata.

For processes and events, *must* transitions are approximated with *may* transitions, yielding an over-approximation. □

The over-approximation is sufficient to prove plan non-existence. In the more simple case where no processes and events are present, the back direction holds as well. For such planning instances I, the translation can also be applied for finding plans because transition sequences in $\mathcal{N}$ are guaranteed to correspond to applicable action sequences in I.

5 Case Study

As a case study, we apply our translation with the SpaceEx model checker (Frehse et al. 2011), which is considered as a state-of-the art tool in the area of hybrid systems model checking. The search engine of SpaceEx performs symbolic search, which is suited for effectively proving plan non-existence. Proving plan non-existence has recently found increasing attention for classical planning (Bäckström, Jonsson, and Ståhlberg 2013), and becomes even harder for planning in hybrid domains. For a particular class of planning domains, SpaceEx is guaranteed to find valid plans in solvable domains as well (see below). We consider several instances (with growing size) of the generator (Howey, Long, and Fox 2004) and the car domains (Fox and Long 2006), which are standard and challenging benchmarks in the hybrid planning community. We compare our translation for SpaceEx with the state-of-the-art planners Colin (Coles et al. 2012) and UPMurphi (Della Penna, Magazzeni, and Mercorio 2012). The experiments are performed on an x64 Linux machine with 6 GB of RAM and an Intel i7 CPU (2.20GHz).

The results for unsolvable instances are reported in Table 1. Colin can prove plan non-existence for a restricted class of domains, namely when there is a tight deadline on reaching the goals (which sets a finite horizon for the plan), and each ground action can only be applied a finite number of times. UPMurphi cannot provide any guarantees about plan non-existence as it relies on discretizing the time line and the continuous variables prior to search. In other words, plans might exist for a finer discretization than actually used by UPMurphi. The results for UPMurphi are included in Table 1 for the sake of completeness. We observe that our translation with symbolic search is able to scale better than both Colin and UPMurphi. In particular, our approach is able to effectively prove plan non-existence in the car domain, which is out of scope for both UPMurphi (as discussed) and Colin (as Colin is not able to deal with processes and events, which are present in the car domain).

D	Tool	1	2	3	4	5	6	7	8	9	10
Gen	SpaceEx	0.01	0.09	0.83	4.25	58.61	1214.35	-	-	-	-
Gen	CoLin	0.01	0.1	1.7	32.48	761.28	-	-	-	-	-
Gen	UPMur	0.9	29.42	-	-	-	-	-	-	-	-
Car	SpaceEx	0.98	4.91	9.46	19.65	37.19	59.40	112.43	210.47	350.14	574.71
Car	CoLin	x	x	x	x	x	x	x	x	x	x
Car	UPMur	36.01	445.23	-	-	-	-	-	-	-	-

Table 1: Results in seconds for unsolvable instances. Instance numbers correspond to number of tanks (*generator*) and maximum acceleration (*car*). Abbrev.: '-': tool still running after 30 minutes, 'x': tool cannot handle the problem.

The symbolic search performed by SpaceEx induces an over-approximation of the original system, which is suited for effectively proving plan non-existence. In contrast, applying symbolic search to find plans might result in *spurious* plans, i. e., plans that do not correspond to valid plans in the concrete. However, for the subclass of planning problems that do neither feature processes nor events (according to Prop. 1) and do only include simple differential equations of the form $\dot{x} = c$, the search algorithm by SpaceEx guarantees that a path to a goal corresponds to a valid plan as well. These requirements are satisfied by the generator, but not by the car domain. The results are depicted in Table 2.

Domain	Tool	1	2	3	4	5	6	7	8	9	10
Generator	SpaceEx	0.01	0.03	0.07	0.1	0.19	0.28	0.45	0.65	0.93	1.22
Generator	CoLin	0.01	0.09	0.2	2.52	32.62	600.58	-	-	-	-
Generator	UPMurphi	0.2	18.2	402.34	-	-	-	-	-	-	-
Car	SpaceEx	0.01	0.01	0.01	0.03	0.04	0.05	0.06	0.07	0.08	0.1
Car	CoLin	x	x	x	x	x	x	x	x	x	x
Car	UPMurphi	28.44	386.5	-	-	-	-	-	-	-	-

Table 2: Results in seconds for solvable instances.

Table 2 shows scalability improvements for solvable instances as well. As discussed, the results for the car domain must be taken with care (as the found paths might be spurious), but are included for the sake of completeness. In contrast, for the generator domain, the found paths by SpaceEx are guaranteed to correspond to valid plans. We observe that SpaceEx outperforms the other tools by several orders of magnitude in terms of scalability. We remark that our current implementation does not yet extract these plans, but this step is purely technical and efficiently implementable (essentially a call to an SMT solver). Overall, we observe that symbolic search is beneficial for both proving plan non-existence as well as for finding paths to goal states. Generally, symbolic search seems to be well suited for hybrid domains because it handles several paths simultaneously.

6 Conclusions

We have presented a formal translation from PDDL+ to the standard formalism of hybrid automata. Our translation forms the basis for bridging the gap between planning in hybrid domains and model checking of hybrid automata. Our experimental evaluation has shown that the translation can be effectively applied to proving plan non-existence in challenging hybrid domains. In particular, our translation extends the class of tractable planning domains for proving plan non-existence as shown for the car domain. For a particular class of hybrid domains, SpaceEx can also be applied for effectively finding plans. For future research, the precise modeling of *must* transitions in order to avoid spurious plans should be addressed. Furthermore, it will be interesting to apply the translation also with other model checking tools in order to exploit their particular strengths. Generally, we hope that our work forms the basis to eventually allow the planning community to systematically benefit from the large body of research in the area of hybrid automata.

Acknowledgments

We thank Alexander Heinz for the help on the benchmark suite preparation. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS, http://www.avacs.org/).

References

Alur, R.; Courcoubetis, C.; Halbwachs, N.; Henzinger, T.; Ho, P.; Nicolin, X.; Olivero, A.; Sifakis, J.; and Yovine, S. 1995. The algorithmic analysis of hybrid systems. *Theoretical Computer Science* 138:3–34.

Bäckström, C.; Jonsson, P.; and Ståhlberg, S. 2013. Fast detection of unsolvable planning instances using local consistency. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SoCS 2013)*. AAAI Press.

Cimatti, A.; Clarke, E. M.; Giunchiglia, F.; and Roveri, M. 2000. Nusmv: A new symbolic model checker. *STTT* 2(4):410–425.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Strong planning in non-deterministic domains via model checking. In *AIPS*, 36–43.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2012. COLIN: Planning with continuous linear numeric change. *Journal of Artificial Intelligence Research (JAIR)* 44:1–96.

Della Penna, G.; Magazzeni, D.; Mercorio, F.; and Intrigila, B. 2009. UPMurphi: A tool for universal planning on PDDL+ problems. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*. AAAI.

Della Penna, G.; Magazzeni, D.; and Mercorio, F. 2012. A universal planning system for hybrid domains. *Applied Intelligence* 36(4):932–959.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research (JAIR)* 27:235–297.

Fox, M.; Howey, R.; and Long, D. 2005. Validating plans in the context of processes and exogenous events. In *AAAI*, 1151–1156.

Frehse, G.; Guernic, C. L.; Donzé, A.; Cotton, S.; Ray, R.; Lebeltel, O.; Ripado, R.; Girard, A.; Dang, T.; and Maler, O. 2011. Spaceex: Scalable verification of hybrid systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, Lecture Notes in Computer Science, 379–395. Springer.

Frehse, G. 2008. PHAVer: algorithmic verification of hybrid systems past HyTech. *STTT* 10(3):263–279.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Henzinger, T. A. 1996. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, 278–292.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 294–301.

Li, H. X., and Williams, B. C. 2008. Generative planning for hybrid systems based on flow tubes. In *ICAPS*, 206–213.

McDermott, D. V. 2003. Reasoning about autonomous processes in an estimated-regression planner. In *ICAPS*, 143–152.

Penberthy, J. S., and Weld, D. S. 1994. Temporal planning with continuous change. In *AAAI*, 1010–1015.

Shin, J.-A., and Davis, E. 2005. Processes and continuous change in a sat-based planner. *Artif. Intell.* 166(1-2):194–253.