Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-based Systems Group
Prof. Gerhard Lakemeyer, Ph.D.

Master's Thesis

Transforming Robotic Plans with Timed Automata to Solve Temporal Platform Constraints

Tarik Viehmann

December 16, 2019

Advisor: Till Hofmann, M.Sc.
Supervisors: Prof. Gerhard Lakemeyer, Ph.D.,
Prof. Erika Ábrahám, Ph.D.

Acknowledgements

I want thank Till Hofmann, who not only only did an excellent job at supporting me throughout the span of this thesis with helpful discussions, critical feedback and guidance to manage my work, but also got me hooked up with the field of robotics in the first place allowing me to learn and gather a lot of invaluable experience so far and for the time to come.
I also am also very grateful for the discussions with Victor Mataré, who also gave me much appreciated feedback and made me consider many details towards the practical feasibility of my work. I wish to express my sincere thanks to Stafan Schupp for helping me in early stages of this thesis to get ideas on approaches to tackle the tasks i was facing.
I also humbly thank Professor Gerhard Lakemeyer for jointly supervising this thesis with Professor Erika Ábrahám, which I greatly appreciate.
I want to thank the whole Carologistics Team for the awesome experience of competing at the RoboCup in Sydney and Montreal, which also provided me with valuable insights for this thesis.
Finally, I would like to thank my friends and family for their support and encouragement.

Contents

1 Introduction

When using planners to solve robotic tasks, a programmer has to model the domain of interest. This includes the current state of the world, possible actions a robot can perform and the effects of those actions. However, it is often problematic to specify the behavior of a robot based on high-level domain features alone, as lower level components may impose constraints on the executability of high-level actions. As an example we consider a production line with assembling stations operated by autonomous robots, where a product is determined by the order and type of assembling steps it goes through. A robot can pick up materials from and deliver them to the available stations. On an abstract level one could model the domain by treating the stations as black boxes and defining three robot actions: `put`, `pick` and `goto`. Such an abstraction does not include any hardware specifics yet, like the control of an robotic arm or the operation of an attached perception unit that detects the objects to pick up or drop down.

Calls to low-level interfaces are typically hidden from the top-level agent framework and invoked transparently without applying any explicit reasoning. The control flow is asserted from domain actions to the interfaces that control the hardware, e.g, gripper movements to physically grab an object are invoked when the agent decides to pick something up. Responses such as sensor data or control feedback are propagated up to the high-level reasoner that may react based on the gathered input. While often such simple action- and reaction-based connections between the robotic platform and the high-level reasoner suffices, problems occur once the low-level control patterns depend on the domain context.

If in the above production line scenario the robot may have a perception unit that is shut off to save energy and only has to be turned on in advance before performing any grasping task. This cannot be handled within low-level implementations easily without either wasting time at each pick (turning it on when the robot decides to pick, then wait until it is ready, then perform the pick) or wasting energy (turning it on, e.g., while moving when close to the destination, but then it gets turned on when it might not be needed if no pick follows after a move). The essential problem here is that the perception handling is dependent on the context determined by the high-level reasoner.

One option is to lift the low-level control up to the domain of reasoning, such that an agent may actively decide to turn the camera on, when it is deemed necessary. However, this has major drawbacks as identified by Hofmann et al. [46]: Firstly, it increases the domain of reasoning and thus affects the performance when determining the course of action, e.g., by means of a planner. Secondly, mixing hardware-specific control with high-level domain knowledge requires expertise in two different fields of robotics that may be maintained by different people. Any change within either of the layers is bound to affect the respective other one as

well. It is usually desired to decouple the low-level specifics from the abstract domain to gain re-usability for other robots built with different hardware as well as robustness to hardware changes of the current robot.

In general we face the following modeling problem: On the one hand, a programmer of the high-level behavior usually does not want deal with all the low-level constraints when designing the domain, and on the other hand, those imposed constraints may depend on the abstract tasks that are solved. If the low-level specifics are handled at the behavior level, a hardware designer has to adapt the high-level agent code when modifying components. Dealing with those constraints within low-level software is often not feasible as well, since they might require adaptions of- or depend on the agent's plans.

Based on Hofmann et al. [46] it is the goal of this thesis to develop a procedure that allows to model platform specifics separately form the abstract domain, in order to gain modularity and to establish a clear separation of concern, while also accounting for relations between the execution context and the low-level controls. The task is tackled by developing a plan transformation that unifies the different constraints imposed by the platform-specific interfaces on the one side and the high-level reasoning framework on the other side. We are particularly interested in the modeling of precise temporal relations between the two entities, which is why we base our approach on two formalisms that are suitable to model expressive temporal behavior.

Timed automata [4] are considered to specify the control flow of hardware interfaces, allowing to model processes acting upon continuous real-time. In the above example of a control unit for the robot's vision, it may useful to model details, such as minimal time span that has to elapse when the sensors are powered on, in order for them to produce reliable data.
To formulate behavioral patterns of those platform-specific models within a given domain context, we utilize the semantics of the metric temporal logic *t-ESG* [45]. It allows to specify statements that are capable of arguing about the structure of plans, which we utilize in two ways: Firstly, formulas may be defined to postulate the requirements imposed on the executability of domain plans based on low-level specifics. Thereby, a theoretical foundation of the proposed plan transformation is established that is concerned with the extension of domain plans by hardware-specific control patterns that satisfy those requirements. Returning to the control of a perception unit, a formula may state that in advance of any grasping action the camera has to be turned on. A plan $\langle \texttt{goto}, \texttt{pick} \rangle$ that first drives to a location before picking up an object needs to be extended, such that the camera is already powered towards the end of the execution of `goto`.
t-ESG formulas may also be utilized to provide an interface between the reasoning framework and a plan transformation satisfying platform constraints: The top-level agent may express domain-specific concerns regarding the temporal constraints that have to hold in order to prove an abstract plan feasible in the scope of the domain of reasoning. One use case of such a formula could be to express that between `goto` and `pick` there may not be any idle time allowed, such that

a plan transformation cannot force the robot to wait in between those actions in order to turn on the perception unit after the execution of `goto`.

Having platform models defined as timed automata on the one side and an initial abstract plan in the form of an action sequence provided by the high-level reasoner on the other side, together with *t-ESG* formulas in between to guide the platform control during the plan, the different concerns have to be unified in order to determine possible executable plans. We accomplish this by subsequently converting all constraints into the formalism of timed automata with the goal to obtain a single automaton with a designated state that is reachable precisely by those paths inducing transformed plans that satisfy the different concerns. As a starting point, a given abstract plan may be converted into an automaton with a state that models the successful high-level execution of it. This automaton is then iteratively modified, until it also includes all necessary information concerning the platform specifics during execution. The modifications are done in such a way that the possible runs through the system correspond to the different possible plans that are executable according the specified constraints. The problem of finding a transformed plan then equates to utilizing existing model checking tools to solve a reachability task on the constructed automaton. Thereby, the procedure developed within this thesis utilizes similarities that classical planning and scheduling tasks share with model checking tasks of timed automata, hence the resulting approach fits under the umbrella of the well-known *planning via model checking* paradigm [27, 39].

The remainder of this thesis is structured as follows. In Section 2 we introduce the formalism of timed automata, give an overview over model checking tasks and the model checking tool landscape for timed automata and also introduce to the situation calculus variant *t-ESG* as well as its application in `GOLOG` programs. We proceed by presenting other work utilizing similarities between model checking and planning tasks and compare them to our proposed application. Then in Section 3 we outline the responsibilities and restrictions of the proposed plan transformation and present a constraint language as subset of *t-ESG* that may be utilized to formalize the requirements on those transformation tasks. In Chapter 4, the ground work of the preceding section is combined into the development of an encoding procedure that provides an encoding of the proposed plan transformation by the means of a reachability problem on timed automata. The encoded problem can then be solved via existing model checking tools, which can output satisfying traces to answer reachability queries. From those traces, one can synthesize the transformed plans that ensure executability with respect to the encoded constraints, the synthesis is detailed in Chapter 5. We proceed to evaluate the procedure in light of challenges inspired by real-world scenarios in the field of logistics in Chapter 6, before ending with concluding remarks in Chapter 7.

2 Preliminaries

In this section we first introduce classical timed automata as well as some of their extensions to provide a formalism for modeling of low-level specifics. Then we explain the basics of model checking and give an overview over available model checking tools for timed automata. We specifically present the UPPAAL model checker, which is used to gather empirical results of the plan transformation proposed in this thesis. We proceed by providing a introduction to the logic *t-ESG* as formalism to specify the temporal relations between high-level domain knowledge and low-level platform models. *t-ESG* is also suited as backend of high-level agent programs written in GOLOG, as presented to conclude this section. Such an application of *t-ESG* would allow to formulate logical connectives to hardware specifics on the same semantics that a high-level reasoner may operate on, giving a baseline to express dependencies based on specific domain predicates.

2.1 Timed Automata

Timed automata were introduced in [4] as a formalism capable of modeling continuous real time systems. In essence, timed automata are finite automata extended by real-valued variables which model continuous time. These variables are called *clocks* and get initialized with 0 when the system starts. They update their time synchronously and at the same rate. To model time-dependent behavior, transitions are extended by *guards*, which are constraints based on the clocks. A transition may only be taken if the current clock values satisfy the associated guard, such transitions are called *enabled*. Upon taking an enabled transition some clocks may get reset to 0, these resets are also referenced as *updates*.
It is not required to take a transition once it is enabled. Instead, there are different approaches to model the need of a system to progress. In [4] progression was ensured by defining Büchi acceptance conditions, meaning a system run is only valid if it visits certain states infinitely often (thus leaving them infinitely often as well). Henzinger et al. introduced a different approach called timed safety automata in [41], which instead allowed to formulate invariant conditions on states that have to be met in order to visit or stay in said state.

2.1.1 Syntax

The notation in the remaining sections is based on timed safety automata and adapted from [15], because this thesis aims to utilize timed automata to model low-level specifics that should ensure the executability of high-level actions, which

is typically determined by locally satisfying some preconditions that in term should ensure a successful execution at that given time. The local view on progress offered through timed safety automata seem to go along naturally with the local scope that actions typically have within plans: They are concerned with the preconditions upon execution, everything prior to the execution start and after effects the effects are applied are no concern for an action, hence we side with this formalism.

Definition 2.1.1 ([15]). *A* timed automaton *can be formally defined as a quadruple* $\mathcal{A} = (L, l_0, E, I)$ *over a finite set of clocks* $\mathcal{C}$ *and a finite alphabet* Σ*, where*

- L *is a finite set of locations (also called states)*
- $l_0 \in L$ *is a starting location*
- $E \subseteq L \times \Phi(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times L$ *is the set of transitions.*

  $\Phi(\mathcal{C})$ *denotes the set of* clock constraints δ *(also called guards), which can be defined inductively via*

  $$\delta ::= x \ \texttt{op} \ c \mid c \ \texttt{op} \ x \mid \neg\delta \mid \delta \wedge \delta$$

  where $x \in \mathcal{C}$ *is a clock,* $\texttt{op} \in \{\leq, \geq\}$ *and* $c \in \mathbb{Q}$ *is a constant.*

  Instead of $(l_i, g, a, r, l_j) \in E$ *we write* $l_i \xrightarrow{g,a,r} l_j \in E$ *and call* r *the set of* resets *or* updates.
- $I : L \to \Phi(\mathcal{C})$ *describes for each location the associated invariant.*

An exemplary timed automaton modeling a perception unit as described earlier is shown in Figure 2.1.

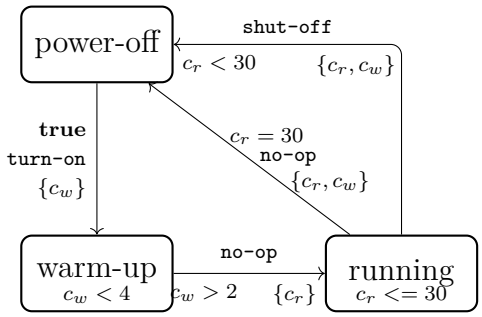


Figure 2.1: TA to model the perception unit, it takes 2-4 time units to warm up and can run up to 30 time unit at which point it automatically turns itself off, it uses two clocks c_w and c_r to count the time since the perception last started to warm up or to run, respectively.

For the remainder of this thesis, we consider timed automata with the following restrictions:

- Constants in clock constraints are from the domain $\mathbb{N}$ instead of $\mathbb{Q}$.

  Since it is easy to scale all clock constraints such that constants from $\mathbb{Q}$ are shifted to a value in $\mathbb{N}$, this basically equates to changing the unit of time measurement, without affecting the timed automaton model.

- Clock constants do not contain disjunctions, hence they also do not contain negations of conjunctions.

  Each clock constraint γ containing disjunctions may be transformed into disjunctive normal form and in case γ appears as guard of a transition, said transition can be replaced by multiple copies, each having a conjunctive clause of γ as guard. If γ describes a location invariant, then said state may be split into multiple ones along the conjunctive clauses if γ. Therefore, the expressive power of the formalism is not restricted by this assumption.

- Clock Constraints may contain *difference constraints*.

  A difference constraint, also called *diagonal constraint* has the form $x-y$ `op` c, where $x, y \in \mathcal{C}$ are clocks, `op` $\in \{<, >, \leq, \geq, =\}$ and $c \in \mathbb{Q}$ is a constant. This extension of the classical formalism does not impact the expressive power of timed automata, as shown in [22].

2.1.2 Semantics

Clock assignments are used to express the current value of each clock. Formally a clock assignment is a function $\nu : \mathcal{C} \to \mathbb{R}$ that assigns time values (from the domain $\mathbb{R}$) to each clock. In the following we introduce notations that allow to easily express changes to clock assignments: Given $t \in \mathbb{R}$, let $\nu + t$ denote the clock assignment that maps any clock $x \in \mathcal{C}$ to $\nu(x) + t$, similarly $\nu \cdot t$ is given by $X \mapsto t \cdot \nu(x)$ and given $U \subset \mathcal{C}$ we write $[U \mapsto t]\nu$ to describe a clock assignment which agrees on all clocks except the ones in U with ν and maps all clocks in U to the value t.
The semantics of a timed automaton $\mathcal{A}$ can be expressed by a transition system $\mathfrak{T}(\mathcal{A}) = (V_{\mathfrak{T}}, E_{\mathfrak{T}})$. $V_{\mathfrak{T}}$ is given by pairs $\langle l, \nu \rangle \in V_{\mathfrak{T}}$ where $l \in L$ is a location and ν is a clock assignment. Since clocks are real-valued, $\mathfrak{T}(\mathcal{A})$ has infinitely many states. To define $E_{\mathfrak{T}}$ we first note that $e = l_i \xrightarrow{g,a,r} l_j \in E$ describes the transition from location l_i to l_j, which can be taken if the clocks satisfy the guards g. If the transition is taken then all clocks in r update their values to 0. At any time a system may either wait in its current location or take a transition from $\mathcal{A}$. When defining $E_{\mathfrak{T}}$ we therefore distinguish between two types of transitions in $\mathfrak{T}(\mathcal{A})$, delay transitions and action transitions:

1. $\langle l, \nu \rangle \xrightarrow{d} \langle l, \nu + d \rangle$, if $\nu + d$ satisfies $I(l)$ for any non-negative real $d \in \mathbb{R}_+$
2. $\langle l, \nu \rangle \xrightarrow{a} \langle l', \nu' \rangle$, if $l \xrightarrow{g,a,r} l' \in E$, ν satisfies $g, \nu' = [r \mapsto 0]\nu$ and ν' satisfies $I(l')$

Figure 2.2 shows part of the transition system of the automaton in Figure 2.1. Beginning from the state `power-off` and with both clocks having the value 0, transitions from `power-off` to `warm-up` are shown. Action transitions are drawn vertically, delay transitions are shown as horizontal arcs.

Executions of processes modeled through timed automata can be expressed via

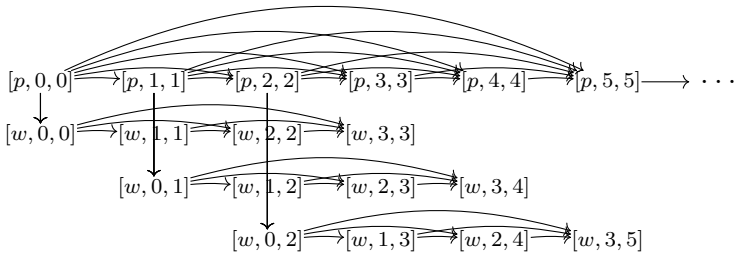


Figure 2.2: A part of the transition system of the automaton in Figure 2.1. Only integer steps in clocks shown, State $[x, y, z]$ represents state x of the TA with clock assignment $c_w = y, c_r = z$, state names are shortened to first letter.

timed traces. A timed trace is a (possibly infinite) sequence of *timed actions* $\langle a_i, t_i \rangle$ with $t_i \leq t_{i+1}$ for all $i \geq 1$. A timed action is a pair $\langle a, t \rangle \in \Sigma \times \mathbb{R}$ and denotes that a is taken after t time steps since $\mathcal{A}$ started. A *run* of $\mathcal{A}$ over a timed trace ξ is a sequence of transitions on $\mathfrak{T}(\mathcal{A})$ that induces ξ. The *timed language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all timed traces ξ for which there exists a run of $\mathcal{A}$ over ξ. One has to be aware that the general notion of timed traces allow for *Zeno traces*, meaning traces that are infinite and time converging, which would require to execute infinitely many actions in finite time. For practical applications it therefore make sense to only consider finite and infinite non-Zeno traces.

2.1.3 Region Graphs

A major reason for the relevance of timed automata as model for real time processes stems from the observation that $\mathfrak{T}(\mathcal{A})$ can represented using finite models. This is due to the low resolution of constants from domain $\mathbb{Q}$ or $\mathbb{N}$ compared to the domain $\mathbb{R}$ of clock values. The concept behind this is, that while time flows continuously and therefore currently elapsed time may only be expressible by irregular numbers, the means to capture the current time are less precise, e.g., stopping the time after exactly π seconds is impossible. To evaluate comparisons within clock constraints the current time has to be captured, hence the value to compare against suffices to have low resolution. Alur and Dill utilized this to define the notion of region graphs [4].

Definition 2.1.2 (adapted from [15])**.** *Let $\mathcal{A} = (L, l_0, E, I)$ be a timed automaton with clocks $\mathcal{C}$. Let $k : \mathcal{C} \to \mathbb{N}$ be a function called* clock ceiling*. For $r \in \mathbb{R}$ let $\lfloor r \rfloor$ denote its integral part and $\{d\}$ denote its fractional part. The* equivalence relation *$\sim_k$ is defined over the set of all clock assignments. $\mu \sim_k \nu$ iff all the following holds:*

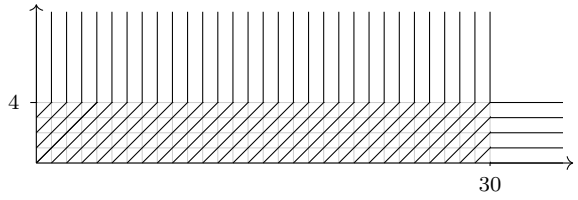
- *For all $x \in \mathcal{C}$: either $\lfloor \mu(x) \rfloor = \lfloor \nu(x) \rfloor$, or $\mu(x) > k(x)$ and $\nu(x) > k(x)$.*
- *For all $x, y \in \mathcal{C}$ with $\mu(x) \leq k(x)$ and $\nu(x) \leq k(x)$: $(\{\mu(x)\} < \{\mu(y)\}$ iff $(\{\nu(x)\} < \{\nu(y)\}$.*

- *For all* $x \in \mathcal{C}$ *with* $\mu(x) \leq k(x)$*:* $(\{\mu(x)\} = 0$ *iff* $\{\nu(x)\} = 0$*.*

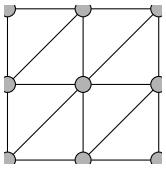
The *region graph* $\mathfrak{R}_k(\mathcal{A})$ can be defined based on $\mathfrak{T}(\mathcal{A})$ using the equivalence classes $[\cdot]$ given through $\sim_k$ via:

1. $\langle l, [\nu]\rangle \xrightarrow{d} \langle l, [\mu]\rangle$, if $\langle l, \nu\rangle \xrightarrow{d} \langle l, \mu\rangle$
2. $\langle l, [\nu]\rangle \xrightarrow{a} \langle l', [\mu]\rangle$, if $\langle l, \nu\rangle \xrightarrow{a} \langle l', \mu\rangle$

The regions essentially utilize the fact that, despite clocks having values from the domain $\mathbb{R}$, the values to compare them against are natural numbers. Therefore, in between two natural numbers the concrete values of clocks does not matter, instead only the relative ordering of clock values is relevant to the evaluation of clock constraints. Locations $s \in V_{\mathfrak{T}}$ that are grouped to a region satisfy the same clock constraints. Beyond a certain clock ceiling, that can be determined by finding the maximal constants that a clock is compared against, the further progress of a clock value is not relevant anymore. Instead, there is again only a distinction based on the relative orderings of clock values.



(a) All regions, ceilings are $c_r = 30$ and $c_w = 4$.



(b) 33 regions from a 2×2 grid with both clocks below their ceilings.

Figure 2.3: Visualization of regions for the automaton in Figure 2.1

Figure 2.3 visualizes the regions of the automaton of Figure 2.1 with clock ceilings $k(c_w) = 4$ and $k(c_r) = 30$. Even tho the timed automaton only has two clocks there are 860 possible regions for each location, resulting in a representation that, despite being finite, is still too big to handle in practical applications.

2.1.4 Zone Graphs

A smaller partitioning of $\mathfrak{T}(A)$ can be obtained by considering *zones* [33] instead of regions. Zones are representations of solution sets of clock constraints. They can be efficiently stored in *difference bound matrices* (DBMs). DBMs are square matrices that have *bounds* $b \in \mathbb{Z} \times \{<, \leq\} \cup \langle\infty, <\rangle$ as values. Entry $[i, j] = \langle c, \texttt{op}\rangle$ encodes a constraint $x_i - x_j$ op c. The following observation provides the basis of DBMs in context of timed automata.

Observation 2.1.3 (see [33])**.** *Solution sets of clock constraints can be represented by bounds on individual clocks together with bounds on differences between pairs of clocks.*

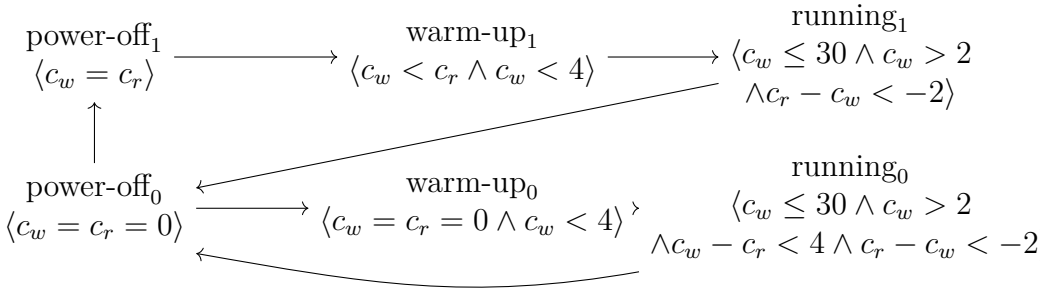


Figure 2.4: Zone graph for the automaton of Figure 2.1.

Note how bounds on individual clocks can be represented as difference bounds as well, when utilizing a virtual clock $\mathtt{cl}_0$ that has a constant value of 0: For $c \in \mathcal{C}, x, y \in \mathbb{N}$ the Bounds $x \ \mathtt{op}_1 \ c \ \mathtt{op}_2 \ y$ may be expressed as

$$c - \mathtt{cl}_0 \ \mathtt{op}_2 \ y \wedge \mathtt{cl}_0 - c \ \mathtt{op}_1 \ -x$$

Therefore, a zone may be represented via a $(|\mathcal{C}|+1)\times(|\mathcal{C}|+1)$ DBM. In the following we refer to zones as sets of clock constraints as well as the set of clock assignment satisfying the constraints interchangeably. Let D be a zone and r be a set of clocks, then one can define the delay of a zone via $D^{\uparrow} := \{\nu + d \mid \nu \in D, d \in \mathbb{R}_+\}$ and the effect of clock resets as $r(D) := \{[r \to 0]\nu \mid \nu \in D\}$ and therefore define symbolic semantics via:

- $\langle l, D \rangle \rightsquigarrow \langle l, D^{\uparrow} \wedge I(l) \rangle$
- $\langle l, D \rangle \rightsquigarrow \langle l, r(D \wedge g) \wedge I(l') \rangle$, if $l \xrightarrow{g,a,r} l'$.

Starting from an initial zone where all clocks are exactly 0 the resulting transition system replicates the semantics of $\mathfrak{T}(\mathcal{A})$, however it is not necessarily finite. Indeed it is easy to find examples, where this is not the case, e.g., shown in [15]. The missing piece is that the maximal constants can be utilized to ensure a finite system. Similar to the clock ceiling function for region equivalence one can define functions to unify different zones that only differ in clock values above the limits. Given such a ceiling function $\mathtt{norm}$ for zones a finite *zone graph* $\mathfrak{G}(\mathcal{A})$ is defined via:

- $\langle l, D \rangle \rightsquigarrow \langle l, \mathtt{norm}(D^{\uparrow}) \wedge I(l) \rangle$
- $\langle l, D \rangle \rightsquigarrow \langle l, \mathtt{norm}(r(D \wedge g) \wedge I(l')) \rangle$, if $l \xrightarrow{g,a,r} l'$.

In case of *diagonal-free* timed automata (timed automata without difference constraints), one may simply define such a ceiling via: $\mathtt{norm}_k(D) := \{u \mid u \sim_k v, v \in D\}$. In the general case this is not possible, but a modified ceiling function can be constructed as shown in [14]. Figure 2.4 shows the zone graph of the automaton from Figure 2.1 having only six states.

In order to get actual information about a timed automaton $\mathcal{A}$ when basing algorithms on the corresponding zone automaton $\mathfrak{G}(\mathcal{A})$, the following concepts are essential:

Definition 2.1.4. *A* symbolic trace *for a timed automaton* $\mathcal{A} = (L, l_0, E, I)$ *is a sequence*

$$\langle l_1, D_1\rangle \overset{g_1,a_1,r_1}{\rightsquigarrow} \langle l_2, D_2\rangle \overset{g_2,a_2,r_2}{\rightsquigarrow} \dots \overset{g_n,a_2,r_n}{\rightsquigarrow} \langle l_n, D_n\rangle,$$

such that $\emptyset \subset D_n \subseteq I(l_n)$ *and for all* $i < n$

- $D_i \neq \emptyset$,
- $D_i \subseteq I(l_i)$,
- $D_i \wedge g_i \neq \emptyset$,
- *there exists* $l_i \xrightarrow{g_i,a_i,r_i} l_{i+1} \in E$ *and*
- $r_i(D_i \wedge g_i) \cap I(l_{i+1}) \subseteq D_{i+1}$.

The above definition essentially requires a symbolic trace to always contain at least one timed trace of $\mathcal{A}$. The following two properties, as proposed in [68], aim to relate symbolic traces even closer to timed traces.

Definition 2.1.5. *A symbolic trace* $\langle l_1, D_1\rangle \overset{g_1,a_1,r_1}{\rightsquigarrow} \langle l_2, D_2\rangle \overset{g_2,a_2,r_2}{\rightsquigarrow} \dots \overset{g_n,a_n,r_n}{\rightsquigarrow} \langle l_n, D_n\rangle$, *is called* post-stable *or* forward-stable*, if for all* $v \in D_i, 1 < i \leq n$*, there exist* $u \in D_{i-1}$ *and* $d \in \mathbb{R}_{\geq 0}$*, such that* $u + d \models g_{i-1}$ *and* $[r_{i-1} \mapsto 0](u + d) \models I(l_i)$.

Definition 2.1.6. *A symbolic trace* $\langle l_1, D_1\rangle \overset{g_1,a_1,r_1}{\rightsquigarrow} \langle l_2, D_2\rangle \overset{g_2,a_2,r_2}{\rightsquigarrow} \dots \overset{g_n,a_2,r_n}{\rightsquigarrow} \langle l_n, D_n\rangle$, *is called* pre-stable *or* backward-stable*, if for all* $u \in D_i, i < n$*, there exist* $d \in \mathbb{R}_{\geq 0}$*, such that* $u + d \models g_i$*,* $u + d \in D_i$ *and* $[r_i \mapsto 0](u + d) \in D_{i+1}$.

A post-stable symbolic trace therefore guarantees that any concrete state in a zone can be reached from one state contained in the predecessor zone, while a pre-stable symbolic trace guarantees that from any concrete state in a zone one may reach a state in the successor zone. The extraction of a concrete trace from a given pre- and post-stable trace is presented in Section 5.

2.1.5 Difference Bound Matrices

The set of bounds together with:

- $\mathfrak{n} := \langle \infty, < \rangle$ (neutral element of $\cap$)
- $\mathfrak{e} := \langle 0, \leq \rangle$ (neutral element of $+$)
- $\langle x_1, \mathtt{op}_1\rangle + \langle x_2, \mathtt{op}_2\rangle := \langle x_1 + x_2, \min\{\mathtt{op}_1, \mathtt{op}_2\}\rangle$, with $<$ being smaller than $\leq$ and $\infty + x = x + \infty := \infty$
- $\langle x_1, \mathtt{op}_1\rangle \cap \langle x_2, \mathtt{op}_2\rangle := \begin{cases} \langle x_1, \mathtt{op}_1\rangle, & \text{if } \langle x_1, \mathtt{op}_1\rangle \leq \langle x_2, \mathtt{op}_2\rangle \\ \langle x_2, \mathtt{op}_2\rangle, & \text{else} \end{cases}$, with $\langle x_1, \mathtt{op}_1\rangle < \langle x_2, \mathtt{op}_2\rangle$ if $x_1 < x_2$ and $\mathtt{op}_1 < \mathtt{op}_2$.

form a *regular algebra* [8, 33], hence DBMs, being square matrices over bounds, form a regular algebra as well. Due to the semantics of DBMs we can immediatly observe the following:

1. For all bounds b in the row corresponding to $\texttt{cl}_0$ it holds: $b \leq \langle 0, \leq \rangle$, because clock values are non-negative.
2. The diagonal entries of all non-empty DBMs are $\langle 0, \leq \rangle$.

Figure Figure 2.5a shows a DBM constructed through the constraints of state running_0 of Figure 2.4. Since no bound was specified for $c_r - \texttt{cl}_0$, the trivial bound $\langle \infty, < \rangle$ is asserted.

	$\texttt{cl}_0$	c_w	c_r
$\texttt{cl}_0$	$\langle 0, \leq \rangle$	$\langle -2, < \rangle$	$\langle 0, \leq \rangle$
c_w	$\langle 30, \leq \rangle$	$\langle 0, \leq \rangle$	$\langle 4, < \rangle$
c_r	$\langle \infty, < \rangle$	$\langle -2, < \rangle$	$\langle 0, \leq \rangle$

(a) Constraints from symbolic state running_0 of Figure 2.4 as DBM

	$\texttt{cl}_0$	c_w	c_r
$\texttt{cl}_0$	$\langle 0, \leq \rangle$	$\langle -2, < \rangle$	$\langle 0, \leq \rangle$
c_w	$\langle 30, \leq \rangle$	$\langle 0, \leq \rangle$	$\langle 4, < \rangle$
c_r	$\langle 28, < \rangle$	$\langle -2, < \rangle$	$\langle 0, \leq \rangle$

(b) Closed form of 2.5a.

Figure 2.5: DBM representations of symbolic state running_0 of Figure 2.4.

However, from

$$c_r - c_w < -2 \Leftrightarrow c_r < -2 + c_w \overset{c_w \leq 30}{\Rightarrow} c_r < 28$$

one can derive that $c_r - \texttt{cl}_0 < 28$. This shows that a solution set of a clock constraint has no unique DBM representation. A canonical representative can be derived by tightening all bounds as much as possible without reducing the solution set:

Definition 2.1.7. *Let D be a DBM. Then $\texttt{cf}(D)$ is called the* closed form *of D, where*

$$\texttt{cf}(D) := \begin{cases} D^* = \bigcap_{i \geq 0} D^i, & \text{if } D \neq \emptyset \\ \emptyset & \text{else} \end{cases}$$

Note that computing the closure $\texttt{cf}(D) = D^*$ of a non-empty DBM D equates to applying the Floyd-Warshall algorithm [35] on the through D induced graph. Testing a DBM D for emptiness can be done by to checking for any self loop with a bound smaller than $\langle 0, \leq \rangle$ after applying the Floyd-Warshall algorithm to D. This is similar to the check for negative cycles after applying the algorithm on a weighted graph, but additionally handles the case of $\langle 0, < \rangle$ in a self loop, which also results in D being empty. A graph representation of the DBM in Figure 2.5a is shown in Figure 2.6. The tightening of the bound $c_r - \texttt{cl}_0 < \infty$ can be derived by taking the path from c_r over c_w to $\texttt{cl}_0$.

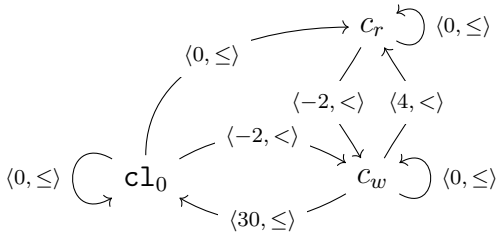


Figure 2.6: Induced Graph of DBM from Figure 2.5a. Trivial Bounds are not shown.

2.1.6 Complexity Results

A first flattening result regarding the complexity that timed automata introduce was found when the language inclusion decision problem was considered. It is the task to decide given two automata $\mathcal{A}$ and $\mathcal{B}$ whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. The initial work in [4] classified this problem as undecidable. It becomes decidable (PSPACE-complete), also due to [4], in case of *deterministic timed automata* where at each state of $\mathfrak{T}(\mathcal{A})$ for each possible action $a \in \Sigma$ at most one transition can be taken. This also implies that non-deterministic timed automata are strictly more expressive than deterministic ones.
A major reason of why timed automata became such a thoroughly studied formalism is due to the complexity of the reachability problem. Said problem is concerned with deciding given two location and clock assignment pairs $\langle l, \nu \rangle, \langle l', \nu' \rangle$ whether there is a path in $\mathfrak{T}(\mathcal{A})$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$. A more general yet in terms of complexity equivalent definition is to give a location l' together with a clock constraint φ as destination. Then the task is to decide if any $\langle l', \nu' \rangle \in \mathfrak{T}(\mathcal{A})$ is reachable such that ν' satisfies φ. Alur and Dill proved that reachability analysis is PSPACE-complete, hence decidable [4]. The proof evolves around the size of region graphs.

State of the art techniques rather utilize smaller symbolic representations, such as DBMs. While reachability analysis may seem unimpressive at first glance, it can be used to check whether failure states can be avoided or safe states are never exited. This technique thus could be used to verify important system properties, such as invariants or safety properties.

2.1.7 Extensions

To further improve the modeling power of timed automata several extensions have been studied. It might become appealing to also allow additive constraints of the form $x + y \ \mathtt{op} \ c$ along with difference constraints. However in [21] it was shown that those, together with difference constraints, makes the reachability problem undecidable for automata with at least four clocks.

Another way of extending the formalism is to add additional freedom when updat-

ing clocks. This has been studied in [19] with the result that the combination of difference constraints and more expressive updates leads to undecidability quickly. Updates of the form $x := y$ (syncing a clock with another clock) or $x :< c$ (updating a clock to a value that is not specified but less than a constant $c \in \mathbb{Q}^+$) can be handled in both cases, the full table of the results in [19] is depicted below in Table 3.1.

Additional updates	$\mathcal{A}_{\text{df}}$	$\mathcal{A}$
$\emptyset$	PSPACE	PSPACE
$\{x := c \mid x \in \mathcal{C}\} \cup \{x := y \mid x, y \in \mathcal{C}\}$	PSPACE	PSPACE
$\{x :< c \mid x \in \mathcal{C}, c \in \mathbb{Q}^+\}$	PSPACE	PSPACE
$\{x := x + 1 \mid x \in \mathcal{C}\}$	PSPACE	Undecidable
$\{x :> c \mid x \in \mathcal{C}, c \in \mathbb{Q}^+\}$	PSPACE	Undecidable
$\{x :> y \mid x, y \in \mathcal{C}\}$	PSPACE	Undecidable
$\{x :< y \mid x, y \in \mathcal{C}\}$	PSPACE	Undecidable
$\{x : \texttt{op}\ y + c \mid x, y \in \mathcal{C}, c \in \mathbb{Q}^+,$ $\texttt{op} \in \{<, \leq, >, \geq, =, \leq\}\}$	PSPACE	Undecidable
$\{x := x - 1 \mid x \in \mathcal{C}\}$	Undecidable	Undecidable

Table 2.1: Impacts on the decidability when strengthening the update capabilities of clocks in timed automata.

Timed automata can also be used to solve optimization problems: In [60] it was shown that reachability tasks can be solved while also returning a minimum time path to the destination. To allow optimizing towards more general objectives [12] introduced *linearly prized timed automata* (PTAs) along with a min-cost reachability algorithm. They allow to define a price function *cost* on locations and transitions. The total price of a run ρ is derived by adding for each visited location l the duration of visits l multiplied by $cost(l)$ and for each transition e $cost(e)$ times the number of times e got used in ρ.

Uncontrollable actions of timed system can be interpreted as a game versus the environment and therefore *timed game automata* (TGAs) [7] have been introduced. They extend timed automata by dividing the transitions into uncontrollable and controllable edges. Time optimal strategies can be synthesized as shown in [6]. PTAs and TGAs can also be combined to form *priced timed game automata* (PTGAs) and used for optimization shown in [18].

2.2 Model Checking

Baier et al. [9] describe the principles of model checking as:

'Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.'

So model checking tasks can be divided into three steps:

1. Model the domain of interest $\mathcal{M}$.
2. Specify a formal property via a logical formula φ.
3. Verify or refute the property by checking whether $\mathcal{M}$ satisfies φ.

Typically if a timed automaton $\mathcal{A} = (L, l_0, E, I)$ is used to model the domain, then a labeling function $\chi : 2^{\mathtt{AP}} \to L$ is introduced along with $\mathcal{A}$ where $\mathtt{AP}$ denotes a finite set of atomic propositions. The labels in $\mathtt{AP}$ are used to describe system properties. This is done to abstract away from the concrete structure of the model and allow to reference properties concerning multiple states. In our application this is not necessary and so we can define $\mathtt{AP}$ such that it consists of unique ids for each location.

2.2.1 TCTL

In order to specify formal properties, one has to define a language capable of expressing them. In our case timed automata are chosen as domain model, which are suited to represent systems with continuous time. Therefore temporal extensions of computation tree logic (CTL) [28] were proposed to express essential real-time system properties. The basis in the form of temporal computation tree logic (TCTL) was given in [2] and the property specification languages of the state-of-the-art model checkers all adapt this formalism to the respective tool features. So in the following we give a definition of TCTL adapted from [9].

Syntax

Given $\mathcal{A} = (L, l_0, E, I)$ with clocks $\mathcal{C}$ with atomic proposition $\mathtt{AP}$ and labeling function χ a TCTL *state formula* can be described as

$$\Psi ::= \text{true} \mid a \mid g \mid \exists\varphi \mid \forall\varphi$$

where $a \in \mathtt{AP}, g \in \Phi(\mathcal{C})$ and φ is a *path formula* defined by $\varphi ::= \Psi \mathbf{U}^I \Psi$ given a convex interval $I \subseteq \mathbb{R}_{\geq 0}$ with integer bounds (including ∞ as upper bound).

Semantics

Let $\text{Path}_{div}(s)$ be the set of all timed traces in $\mathfrak{T}(\mathcal{A})$ starting in $s = \langle l, u \rangle$ that are either finite or non-Zeno. Then the satisfaction relation of TCTL state formulas can be given by

- $s \models a$ iff $a \in \chi(l)$
- $s \models g$ iff u satisfies g
- $s \models \Psi_1 \wedge \Psi_2$ iff $s \models \Psi_1$ and $s \models \Psi_2$

- $s \models \neg\Psi$ iff not $s \models \Psi$
- $s \models \exists\varphi$ iff $\pi \models \varphi$ for some $\pi \in \mathrm{Path}_{div}(s)$
- $s \models \forall\varphi$ iff $\pi \models \varphi$ for all $\pi \in \mathrm{Path}_{div}(s)$

The satisfaction relation for TCTL path formulas can be defined as:

$$\pi = s_0 \to s_1 \to \ldots = \langle\alpha_0, t_0\rangle, \langle\alpha_1, t_1\rangle, \ldots \models \Psi_1 \mathbf{U}^I \Psi_2, \text{ iff } \exists i \geq 0. s_i \models \Psi_2 \text{ and } \forall j < i. s_j \models \Psi_1 \vee \Psi_2 \text{ and } t_i - t_0 \in I$$

For convenience the usual operators from modal logic can be defined as ($\top$ denotes a universally true statement):

- $\Diamond_I \Psi := \top \mathbf{U}_I \Psi$ (there exists s_j where $s_j \models \Psi$ holds and s_j is reached within I)
- $\exists\Box_I \Psi := \neg\forall\Diamond_I\neg\Psi$ (there exists a path where Ψ holds in any state that is reached within I)
- $\forall\Box_I \Psi := \neg\exists\Diamond_I\neg\Psi$ (On all paths Ψ holds in any state that is reached within I)

Most importantly for us is the fact that we can formulate reachability tasks to a location l by introducing an atomic propositions $id(l)$ that is uniquely assigned to l via χ. Then the property $\exists\Diamond_{[0,\infty)}\mathrm{id}(l)$ allows to check whether l is reachable. Modern tools offer to return diagnostic traces which in this case yield a path to l if it exists. To put model checking in context of this thesis, the goal to transform high-level plan according to platform specific requirements is achieved by encoding the problem of finding an executable plan as a reachability problem on timed automata, such that a feasible plan can be obtained from a solution trace.

2.3 Model Checking Tools

We base our work on the model checking tool `UPPAAL` [11], which is jointly developed by Uppsala University and Aalborg University and was first released in 1995.

There are various alternative tools available to solve reachability tasks on timed automata, such as the fully symbolic solver `RED` [69], the well-known command line tool `KRONOS` [20], and also solvers for formalisms that subsume timed automata, e.g. `HYTECH` [42] and `HYPRO` [67] which handle hybrid systems or `FSMTMC` for modeling finite state machines with time [59], to only name a few. Since the input and output formats, as well as the modelling capabilities differ across the tools, it is hard to simply compare them performance-wise in order to justify choosing the use of one tool over the others. Although, effort has been made, e.g. in [23], the general lack of standardized benchmark domains makes it impossible to obtain

competitive rankings, as of yet. The vast differences also imply that it becomes a major task to integrate a single tool for the scope of this thesis, which is why we decided against an attempt to compare different model checkers for our use-case. Instead, the decision in favor of UPPAAL was mainly made due to the following reasons:

- There is detailed documentation available, which tremendously helped to ensure that all required features for a model checking based plan transformation were indeed covered.
- A GUI aallowed to estimate the scalability of our approach early on and was especially helpful to quickly prototype alternative approaches.
- It is actively developed, with the latest development snapshot release from 2019, as of date.

Notable drawbacks of UPPAAL are, that the source code is not publicly available and the provided C++ API, in the form of the companion library UTAPLIB, cannot call the solver directly. Therefore, in order to integrate the functionalities of UPPAAL into a software stack, one has to either use the JAVA API or invoke calls to a companion command line tool verifyta directly. Also, UPPAAL models do not support the notion of action labels on transitions. Since action labels play an important role for the proposed modeling of platform specifics as timed automata, this imposes the following restriction on platform models: any two transitions t_1, t_2 of a platform model automaton $\mathcal{A}$ must differ either in their guards, updates, source or destination locations. In the unlikely case, where this may become an issue, one could add a trivial constraint to the guard to make t_1 and t_2 distinguishable again. This effectively allows us to model timed automata with action labels, handing them without labels to the solver and afterwards retrieve them with a lookup from the original model.

2.3.1 UPPAAL

UPPAAL is written in C++ and comes along with a Java-based GUI. UPPAAL models are based on timed automata with difference constraints saved as .xml files. However, the classical formalism is extended in various ways such as bounded discrete variables, arrays and parameterized automata. Clocks and variables can be defined in either a local (instantiated automaton only) or global (ranging over all automata forming a system) scope. Communication in a system of parallel operating timed automata is realized via channels in a master-slave like relation: A transition can be annotated via a channel identifier c followed by ! or ? to denote an active or passive transition respectively. There are three types of channels: binary, urgent and broadcast which behave as follows given a system of automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ in the current location $(l_1, \ldots, l_n)$.

- Let c be a binary channel. If for some $l_i, i \in [1, n]$ there is a transition e_i annotated with $c!$ and there exists $j \neq i$ s.t. in l_j is a transition e_j annotated

with $c?$ and both guards on e_i and e_j are satisfied as well as the invariants of the target states then the system can take both transitions synchronously.

- Urgent channels behave like binary channels but require the system to take the synchronized edges as soon as possible, therefore guards are not allowed on them.
- Let c be a broadcast channel. If for some $l_i, i \in [1, n]$ there is a transition e_i annotated with $c!$ which guard is satisfied then the system can take this transition along with all transitions e_j annotated with $c?$ from states $l_j, j \neq i$ synchronously (if some l_j has multiple outgoing transitions labeled with $c?$ then only one is used). Transitions annotated with the passive end of a broadcast channel are not allowed to have guards. Also note that this is a non-blocking broadcast in the sense that e_i does not need any receiver to emit the broadcast, however all enabled edges that can receive it will synchronize. This may lead to undefined states if the target locations of the receiving transitions have unsatisfied invariants.

As requirement specification language `UPPAAL` supports a subset of TCTL that does not allow nested quantifications over path formulas. However, they introduce the operator $\rightsquigarrow$ to check for bounded liveness properties of the form $\varphi_1 \rightsquigarrow \varphi_2$, meaning whenever φ_1 holds then eventually φ_2 holds as well, where φ_1 and φ_2 are atomic state formulas.
Based on the success of `UPPAAL` several extensions have been developed to further increase the modeling power, such as `UPPAAL-TIGA` (Timed Interfaces Game Automata) for TGAs [25] and `UPPAAL-CORA` (Cost Optimal Reachability Analysis) [13] for PTAs.

2.3.2 `verifyta`

`UPPAAL` comes along together with the command line tool `verifyta` to allow GUI-less invokations of the solver. In the following we present the basic workflow with `verifyta`: `verifyta` requires an automata system description as `.xml` file together with a query from the supported query language, stored in a file with ending `.q`. It then returns the result of the query and optionally an *symbolic trace* (a sequence of zones together with the connecting automata transitions) satisfying/refuting the query.

We proceed by listing the command line options that are relevant for the scope of this thesis.

- `-t` ⟨`0|1|2`⟩: Generate a trace on stderr. Options 0, 1 and 2 specify the type of trace, any trace, the shortest trace (with respect to the number of transitions taken) or the fastest trace (with respect to total time ellapsed), respectively.

  Using the option `-t 2` we have the basis to extract the time-optimal executable plan as a solution to the proposed plan transformation.

- `-Y`: output a pre- and post-stable trace.

  The possibility to obtain a e pre- and post-stable trace allows to compute a timed trace easily, as shown in Section 5.

- `-f` ⟨`filename`⟩: Write the trace to a file instead of stderr.

The trace output of `verifyta` is written in a format suitable for the `UPPAAL` GUI, but unusuable elsewhere, because internally `verifyta` and `UPPAAL` utilize an intermediate format to store timed automata efficiently and output traces are only containing references to the data of that internal format. However, the `UTAPLIB` comes along with a tool called `tracer`, which can produce a human readable trace given the output trace file of `verifyta` and the intermediate format of the automata system in use. The latter can also be computed by `verifyta` by setting the `COMPILE_ONLY=1` environment variable. The full workflow to obtain a symbolic trace is depicted in Figure 2.7.

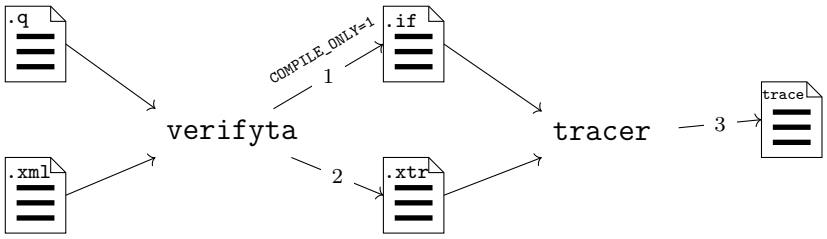


Figure 2.7: Steps to obtain a human readable trace using `verifyta` and `tracer`.

2.4 Metric Temporal Constraints in *t-ESG*

In order to define the constraints that connect modeled platform components with abstract plan actions, we propose to use a formalism that is able to fulfill the following two requirements:

(I) Capability to fully model the high-level domain.

  This enables the formulation of constraints based on the same semantics that the high-level reasoner may operate on. Therefore, a baseline is given to express more general dependencies between platform and domain specifications, rather than just basing platform model interactions on abstract plans only.

(II) Similar expressive power regarding temporal relations between high-level events and platform operations as the timed automata formalism provides.

  The precise temporal modeling of possible platform model operations can only be effectively utilized if the constraints to describe the invocations of said operations allow for a comparable fine-grained temporal control.

To address the first point, some variant of the situation calculus [58, 65], a first-order logic for reasoning about actions, may provide a good basis. A domain is essentially represented by situations and actions that modify them, where situations are snapshots of the world during some execution, containing the knowledge of the finite history of executed actions so far. A domain is defined via a *basic action theory*, a set of logical sentences that can be categorized via the following partitioning:

$$\Sigma \cup \Sigma_0 \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}} \cup \Sigma_{\text{una}},$$

where

- Σ are axioms to attach semantics to sitations,
- Σ_0 describes the initial situation,
- Σ_{pre} formalizes the preconditions of actions,
- Σ_{post} provides effects of actions in the form of *successor state axioms*, sentences that state the exact changes acion executions entail,
- Σ_{una} contains axioms to ensure unique action names.

Several variants of the situation calculus were proposed to enrich the formalism, e.g. the modal variant $\mathcal{ES}$ [53] that avoids the explicit notion of situations and instead pushes axioms about situations and standard names within the language semantics. Claßen and Lakemeyer [29] extended $\mathcal{ES}$ by a notion of time to the logic $\mathcal{ESG}$, which Hofmann and Lakemeyer [45] extended further by metric temporal time and quantitative temporal operators similar to Metric Temporal Logic (MTL)[50].

We base our work on the ideas of [45] and use t-$\mathcal{ESG}$ as language for specifying connective constraints between platform specifics and abstract plans. The remainder of this section summarizes the basic syntax and semantics of t-$\mathcal{ESG}$, adapted from [45]. For more details see [53, 29, 45].

2.4.1 Syntax of t-$\mathcal{ESG}$

t-$\mathcal{ESG}$ is a sorted logic with sorts object, action, and number and a countably infinite set of standard names for each sort, that can be seen as unique identifiers and are isomorphic with the domain of the respective sort.

Definition 2.4.1. *The* symbols of t-$\mathcal{ESG}$ *are:*

- *Object variables* $x_1, x_2, \ldots, y_1, \ldots$ *and object standard names* $\mathcal{N}_O = \{o_1, o_2, \ldots\}$.
- *Action variables* $a_1, a_2, \ldots$ *and action standard names* $\mathcal{N}_A = \{p_1, p_2, \ldots\}$.
- *Number variables* $t_1, t_2, \ldots$ *and number standard names, here we assume* $\mathcal{N}_N = \mathbb{N}$ *in accordance to the allowed constants in clock constraints of timed automata.*
- *Fluent predicates of arity k:* $\mathcal{F}^k = \{f_1^k, f_2^k, \ldots\}$*, with distinguished predicates* $<\in \mathcal{F}^2$ *and* $Poss \in \mathcal{F}^1$.

- *Rigid functions of arity k:* $\mathcal{G}^k = \{g_1^k, g_2^k, \ldots\}$*, with distinguished functions* $+, \cdot \in \mathcal{G}^2$*.*
- *Fluent object functions of arity k:* $\mathcal{H}^k = \{h_1^k, h_2^k, \ldots\}$*.*
- *Fluent number functions of arity k:* $\mathcal{I}^k = \{i_1^k, i_2^k, \ldots\}$*, with distinguished functions time* $\in \mathcal{I}^1$ *and now* $\in \mathcal{I}^0$*.*
- *open, half-open and closed intervals with number constants as endpoints and a possible missing upper bound, denoted via* ∞*).*
- *Connectives and other symbols:* $=, \wedge, \neg, \forall, \boldsymbol{X}_I, \boldsymbol{U}_I, \boldsymbol{V}_I, \boldsymbol{S}_I, \Box, [\cdot], [\![\cdot]\!]$

For a given interval I and constant $c \in \mathcal{N}_N$ we write $c + I$ to denote the interval I' having the bounds of I increased by c, e.g. $2 + (1, 3] = (3, 5]$.

Definition 2.4.2. *The set of t-*$\mathcal{ESG}$ terms *is the least set, such that:*

- *every variable and every standard name is a term of the same sort*
- *given* $t_1, \ldots, t_k$ *and a k-ary function symbol* f*,* $f(t_1, \ldots, t_k)$ *forms a term of the same sort as* f*.*

A term is called primitive*, if it does not contain any variable symbol, we denote the sets of primitive terms of sort object, action and number by* $\mathcal{P}_O, \mathcal{P}_A$*, and* $\mathcal{P}_N$*, respectively.*

Definition 2.4.3. *A t-*$\mathcal{ESG}$ program *is given by the following grammar:*

$$\delta ::= t \mid \alpha? \mid \delta_1; \delta_2 \mid \delta_1 \mid \delta_2 \mid \pi x.\delta \mid \delta_1 \mid\mid \delta_2 \mid \delta^*$$

where t *is an action term and* α *is a static situation formula.*

Similarly to TCTL, where state and path formulas are defined, *t-*$\mathcal{ESG}$ distinguishes between situation and trace formulas.

Definition 2.4.4. *The t-*$\mathcal{ESG}$ situation formulas *are the least set, such that:*

- *If* $t_1, \ldots, t_k$ *are terms and* P *is a k-ary predicate symbol, then* $P(t_1, \ldots, t_k)$ *is a situation formula, if* $t_1, \ldots, t_k$ *are standard names, then* $P(t_1, \ldots, t_k)$ *is called* primitive*.*
- *If* t_1, t_2 *are terms, then* $t_1 = t_2$ *is a situation formula.*
- *If* α, β *are situation formulas,* x *is a variable,* P *a predicate symbol,* δ *is a program and* ϕ *is a trace formula, then* $\alpha \wedge \beta, \neg\alpha, \forall x.\alpha, \Box\alpha, [\delta]\phi$ *and* $[\![\delta]\!]\phi$ *are situation formulas.*

A situation formula is static*, if it does not contain any of the following symbols:* $\Box, [\cdot], [\![\cdot]\!]$*. A situation formula is* fluent*, if it is static and does not contain the predicate Poss. The set of primitive situation formulas is denoted by* $\mathcal{P}_F$*.*

Definition 2.4.5. *The t-*$\mathcal{ESG}$ trace formulas *are the least set, such that:*

- *If α is a situation formula, then it is also a trace formula.*
- *If ϕ, ψ are trace formulas, x is a variable, and I is an interval, then then $\phi \wedge \psi, \neg\phi, \forall x.\phi, \boldsymbol{X}_I\phi, \boldsymbol{V}_I\phi, \phi \boldsymbol{U}_I\psi$ and $\phi \boldsymbol{S}_I\psi$ are trace formulas.*

We also use the following short-hand notations: $\mathbf{F}_I\phi := \top\mathbf{U}_I\phi$ (future), $\mathbf{G}_I\phi := \neg\mathbf{F}_I\neg\phi$ (globally), $\mathbf{P}_I\phi := \top\mathbf{S}_I\phi$ (past), $\mathbf{H}_I\phi := \neg\mathbf{P}_I\neg\phi$ (historically) and if $I = [0, \infty)$, it may be omitted.

2.4.2 Semantics of *t-ESG*

The semantics of *t-ESG* formulas are depending on *worlds* and *timed traces*. Analogously to timed traces in the context of timed automata, in *t-ESG* a timed trace is a sequence of tuples consisting of actions and time points. Timed traces are required to be non-Zeno, the set of finite traces is denoted by $\mathcal{Z}$.

Worlds assign truth values to all predicates for any given finite timed trace (denoting the history of executed actions).

Definition 2.4.6. *A* world *w is defined by mappings*

1. $\mathcal{P}_F \times \mathcal{Z} \to \{0, 1\}$
2. $\mathcal{P}_O \times \mathcal{Z} \to \mathcal{N}_O$
3. $\mathcal{P}_N \times \mathcal{Z} \to \mathcal{N}_N$

For a given world w together with an execution history $z \in \mathcal{Z}$ the semantics of terms, programs and formulas are determined as follows.

Definition 2.4.7. *Let w be a world and $z = \langle (a_1, t_1) \cdot (a_2, t_2) \cdot \ldots \cdot \ldots \cdot (a_n, t_n) \rangle \in \mathcal{Z}$ be a finite trace. The* denotation $|t|^z_w$ of a term t *is defined via:*

- *if $t \in \mathcal{N}_0 \cup \mathcal{N}_A \cup \mathcal{N}_N$, then $|t|^z_w := t$,*
- *if $t = now$, then $|t|^z_w := time(z)$ with $time(z) := t_n$,*
- *if $t = time(a(t_1, \ldots, t_k))$, then $|t|^z_w := \max\{t_a \mid \langle a(n_1, \ldots, n_k), t_a \rangle \in z, n_i = |t_i|^z_w\} \cup \{0\}$,*
- *if $t = f(t_1, \ldots, t_k)$, then $|t|^z_w := w[f(|t_1|^z_w, \ldots, |t_k|^z_w), z]$.*

Note the designated semantics of *now* and *time*, where the former always refers to the current time in the world and the latter holds the time of any actions' last execution. Since worlds determine the values of functions, objects and numbers based on a history that contains time points, it is possible to model complex temporal relations, such as fluent predicates that are not only determined by the sequence of executed actions from an initial world state, but also depend of the timings of the executed actions. As an example, given a predicate $Holding(\vec{x})$ and an action *pick*, then $w[Holding(\vec{x}), \langle (pick, 10) \rangle]$ could be 1 while $w[Holding(\vec{x}), \langle (pick, 20) \rangle]$

evaluates to 0 in the same world, e.g., because the object to pick up is moving on a conveyor belt causing it to be pick-able only in certain time frames.

Semantics of programs are defined via transitions between configurations, the semantics are sketched informally as follows:

- $\alpha?$ denotes test,
- $\delta_1; \delta_2$ a sequence,
- $\delta_1 \mid \delta_2$ non-deterministic branching,
- $\pi x.\delta$ non-deterministic choice of arguments,
- $\delta_1 \mid\mid \delta_2$ interleaved concurrency
- and δ^* denotes non-deterministic iteration.

The formal definitions are specified in [45]. Relevant for the scope of this paper is just, that programs induce a set of timed traces $||\delta||_w^z$ consisting of all finite successful executions of δ in w when executed after history z, as well as all infinite traces that lead a program to never terminate. This is sufficient to formally define the semantics of formulas.

Definition 2.4.8. *Let w be a world and $z \in \mathcal{Z}$ be a finite trace. The* truth of a situation formula *is defined inductively by:*

- $w, z \models F(t_1, \ldots, t_k)$*, iff* $w[F(|t_1|_w^z, \ldots, |t_k|_w^z)] = 1$,
- $w, z \models (t_1 = t_2)$*, iff* $|t_1|_w^z$ *and* $|t_2|_w^z$ *are identical,*
- $w, z \models \alpha \wedge \beta$*, iff* $w, z \models \alpha$ *and* $w, z \models \beta$,
- $w, z \models \neg\alpha$*, iff* $w, z \not\models \alpha$,
- $w, z \models \forall x.\alpha$*, iff* $w, z \models \alpha_n^x$ *for all n of the same sort as x, where α_n^x refers to α with all occurrences of x replaced by n,*
- $w, z \models \Box\alpha$*, iff* $w, z \cdot z' \models \alpha$ *for all* $z' \in \mathcal{Z}$,
- $w, z \models [\delta]\alpha$*, iff* $w, z \cdot z' \models \alpha$ *for all finite* $z' \in ||\delta||_w^z$,
- $w, z \models [\![\delta]\!]\phi$*, iff* $w, z, \tau \models \phi$ *for all* $\tau \in ||\delta||_w^z$.

So, intuitively $\Box\alpha$ denotes that α holds true independent from any future actions that may be executed, $[\delta]\alpha$ means that α holds true after every successful execution of δ and $[\![\delta]\!]\phi$ holds true if ϕ is satisfied during every possible execution of δ.

Definition 2.4.9. *Let w be a world, $z \in \mathcal{Z}$ be a finite trace (the history) and τ a (possible infinite) trace (the remaining sequence to execute), such that $z \cdot \tau$ forms a trace. The* truth of a trace formula *is defined by:*

- $w, z, \tau \models \alpha$*, iff* $w, z \models \alpha$ *and α is a situation formula,*
- $w, z, \tau \models \phi \wedge \psi$*, iff* $w, z \models \phi$ *and* $w, z \models \psi$,
- $w, z, \tau \models \neg\phi$*, iff* $w, z \not\models \phi$,

- $w, z, \tau \models \forall x.\phi$, *iff* $w, z \models \phi_n^x$ *for all* n *of the same sort as* x, *where* ϕ_n^x *refers to* ϕ *with all occurrences of* x *replaced by* n,
- $w, z, \tau \models \boldsymbol{X}_I\phi$, *iff there exists* $p \in \mathcal{P}_A$ *with* $\tau = p \cdot \tau'$, $w, z \cdot p, \tau' \models \phi$ *and* $time(p) \in time(z) + I$,
- $w, z, \tau \models \boldsymbol{V}_I\phi$, *iff there exists* $p \in \mathcal{P}_A$ *with* $z = z' \cdot p$, $w, z', p \cdot \tau' \models \phi$ *and* $time(p) \in time(z') + I$,
- $w, z, \tau \models \phi \boldsymbol{U}_I\psi$, *iff there exists* $z_1 \in \mathcal{Z}$ *with:*
  - $\tau = z_1 \cdot \tau'$,
  - $time(z \cdot z_1) \in time(z) + I$,
  - $w, z \cdot z_1, \tau' \models \psi$
  - *for all* $z_2 \neq z_1$ *with* $z_1 = z_2 \cdot z'$ *for some* $z' \in Z$, $w, z \cdot z_2, z' \cdot \tau' \models \phi$
- $w, z, \tau \models \phi \boldsymbol{S}_I\psi$, *iff there exists* $z_1 \in \mathcal{Z}$ *with:*
  - $z = z_1 \cdot z_2$,
  - $time(z) \in time(z_1) + I$,
  - $w, z_1, t_2 \cdot \tau \models \psi$
  - *for all* $z_3 \neq \langle\rangle$ *with* $z_2 = z_3 \cdot z'$ *for some* $z' \in Z$, $w, z \cdot z_3, z' \cdot \tau \models \phi$

We briefly give the semantics of the trace operators on an intuitive level:

- $\mathbf{X}_I\phi$ holds iff the next state (e.g. the next action in the program) is reached within a time window of I and in that state ϕ holds.
- $\mathbf{V}_I\phi$ holds iff in the previous state (e.g. the preceding action in the program) ϕ holds and the current state was reached within a time window of I from that state.
- $\phi\mathbf{U}_I\psi$ holds iff there is a future state reached within a time window of I and in that state ψ holds. Additionally on all intermediate state ϕ must be true (without this last condition we obtain the semantics of $\mathbf{F}_I$). So the t-$\mathcal{ESG}$ semantics of $\mathbf{U}_I$ correspond to those of TCTL path formulas.
- $\phi\mathbf{S}_I\psi$ holds iff there is a past state where ψ holds and from which the current one is reached within a time window of I. Additionally on all intermediate states ϕ must be true (without this last condition we obtain the semantics of $\mathbf{P}_I$).
- $\mathbf{G}_I\phi$ holds iff in all future states within I ϕ holds.
- $\mathbf{H}_I\phi$ holds iff in all past states from which the current one is reachable within I ϕ holds.

Similar to classical situation calculus, t-$\mathcal{ESG}$ can be used to express basic action theories. Due to the embedding of standard names and situations (in t-$\mathcal{ESG}$ given in form of worlds and associated traces) in the semantics, the formulation of a basic

action theory is significantly easier, a feature inherited from $\mathcal{ES}$. In particular, it is sufficient to provide a description of the initial world, all preconditions and the successor state axioms:

Definition 2.4.10. *Let $\mathcal{F}$ be a set of fluent predicates and Σ be a set of t-$\mathcal{ESG}$ sentences. Σ is called a* basic action theory over $\mathcal{F}$ (BAT)*, iff $\Sigma = \Sigma_0 \cup \Sigma_{pre} \cup \Sigma_{post}$, where Σ only mentions fluents from $\mathcal{F}$ and*

- *Σ_0 is any set of fluent sentences,*
- *Σ_{pre} is a set of fluent formulas with free variable a.*
- *Σ_{post} is a set of sentences containing exactly the following formulas:*
  - *for each $f \in \mathcal{F} : \Box[a]f(\vec{x}) \equiv \gamma_f$ and*
  - *for each $f' \in \mathcal{H} \cup \mathcal{I}$: $\Box[a]f'(\vec{x}) = v \equiv \gamma_{f'}$,*

  where γ_f and $\gamma_{f'}$ are fluent formulas and v is a variable.

A single precondition axiom is formed from Σ_{pre} via $\Box Poss(a) \bigvee \Sigma_{\text{pre}}$.

For a full BAT specified in t-$\mathcal{ESG}$ we refer to [45], in the following we provide example sentences for each of the three required types of sets. By considering a robot operating in a room with a $goto(s, d)$ action and fluent predicates $At(r, l)$ specifying the robots location as well as Occ referring to the currently started action.

- $At(\text{TABLE}) \in \Sigma_0$ may describe that the robot is initially at a table,
- $\exists s \exists d.a = goto(s, d) \wedge At(s) \in \Sigma_{\text{pre}}$ provides the preconditions of $goto$, in this case the only requirement is that the robot has to be at the source location.
- $\Box[a]Occ(a') \equiv a = a' \in \Sigma_{\text{post}}$ requires Occ to always point to the currently started action,

  $\Box[a]At(l) \equiv a = goto(s, l) \vee (At(l) \wedge a \neq goto(l, d)) \in \Sigma_{\text{post}}$ states that the location of the robot changes only due to an executed $goto$ action.

2.5 Agent Programming in GOLOG

GOLOG [55] is a language suitable for high-level agent programming. The core idea behind it is to formulate the agent's capabilities (possible actions and the current world state) by means of a basic action theory in the situation calculus and provide macros to formulate programs defining the agents behavior. Those macros include both deterministic control structures such as `if` conditionals or `while` loops and non-deterministic aspects such as choice of actions, parameters or number of iterations. This allows to mix hand-crafted behavior with dynamic decisions. The used macros can be expanded back to formulas from the situation calculus in such a way, that suitable courses of action can be determined by applying theorem

provers to the expanded program together with the knowledge about the current situation.
Basically, Levesque et al. [55] added program semantics to the situation calculus to enable high-level specifications of agent behavior based on models defined by logical formulas.

The idea is preserved in *t-ESG* with the available program semantics. Therefore, agents written in GOLOG with *t-ESG* as logical back end aligns well with the idea to have a high-level reasoner that operates on the same semantics as the connective constraints expressing low-level requirements. The program semantics of *t-ESG* can be used directly to define the common GOLOG macros:

- **while** ϕ **do** δ **done** $:= (\phi?;\delta)^*\neg\phi?$
- **if** ϕ **then** δ_1 **else** δ_2 **fi** $:= [\phi?;\delta_1]|[\neg\phi?;\delta_2]$
- **if** ϕ **then** δ **fi** := **if** ϕ **then** δ **else** `nil` **fi**

2.5.1 golog++

Despite the flexibility to combine planning and iterative programming approaches that high-level agent programmers gain when using GOLOG, it is safe to say, that as of date the usage of classical planning approaches is prevalent. Matare et al. [57] argue that an important factor to this is not due to shortcomings of the GOLOG formalism, but rather due to more practical reasons. Classical implementations of GOLOG are written in the logic programming language prolog and often lack important engineering features such as consistency checks, comprehensive debug output and documentation. As solution [57] presents golog++, a GOLOG interfacing framework with three major concerns:

1. Representation of GOLOG programs by providing a syntax for specifying GOLOG programs that is parsed in a templated C++ class model.

   This not only provides a well defined metamodel for GOLOG programs with consistency checks, but also yields an interface to use when integrating into a robotic software stack.

2. Static and runtime semantics that are attachable to the programs instantiated object model.

   The implementation of GOLOG semantics is not specified, but rather wrapped and applied to the metamodel, giving an interface to the logic backend.

3. Defined interfaces for acting and sensing.

   A resulting GOLOG program has to be executed by some platform backend and feedback has to be injected back to the metamodel to capture exogenous events and the gathering of sensor data.

We are currently working towards an integration of the procedures presented in this

thesis into the golog++ framework to eventually apply it to real-world scenarios.

2.6 Related Work

While related work to decouple low-level specifics from the abstract domain was presented in [46], here we focus on work that utilizes the strong connection between model checking and planning tasks and also present applications utilizing timed automata in related problem settings.

2.6.1 Model Checking and Planning

The idea of using model checkers to solve planning related task is quite common due to the similarities of the tasks, as outlined in the following.

The classical formulation of a planning problem states that given an *initial state* one shall reach a *goal state* [66]. In order to solve such a planning instance one can divide the procedure into three steps [39], highlighting the similarities to the model checking procedure in Section 2.2:

1. Model the domain with the possible world states, available actions and state transitions caused by the execution of actions.
2. Define a planning problem by specifying an initial state and a set of goal states.
3. Generate a plan by exploring the state space starting in the initial state. At each step logical formulas over the planning domain are evaluated to determine suitable next actions.

Various approaches have been developed to transform planning domains into semantic models and goal specifications into suiting formal system properties under the paradigm *planning via model checking*. Similarly there is also active research to aid model checking procedures with the help of advances in the field of planning, known as the *directed model checking* paradigm [27].

The plan transformation approach presented in this thesis fits the planning via model checking paradigm, as domain details are decoupled from the abstract planning process and instead integrated afterwards using model checking formalisms and tools.

Planning via Model Checking

The potential of model checking tools applied directly to planning domains were studied for example in [56], where the performance of different model checking tools was measured.

Firstly, a time-bounded planning problem, the bridge-crossing puzzle, was taken with the task to find any solution that lies within a given upper bound. Three model checking tools, namely `NuSMV` [26], `SPIN` [47] and the presented tool `PAT`, competed against the planning system `Metric-ff` [43] with the result that, although being overall slightly worse, `SPIN` and `PAT` were scaling comparably to `Metric-ff` on increasing problem sizes.
Secondly, a deterministic optimal planning domain was considered, the sliding puzzle game, and the three model checkers were compared against the performance of `SATPlan` [48], with the result that `SPIN` and `PAT` both outperformed `SATPlan` on difficult instances, as their performance was relatively constant across the considered difficulties, whereas `SATPLAN` already showed a rapid scaling of runtime with increasing solution length. However, solutions calculated via `SPIN` were not optimal.

Research also went into the exploiting of similarities between classifications planning problem specifications, such as the different PDDL specifications [38], and the model checking framework of SAT Modulo Theories [32] (SMT). There, boolean skeleton formulas containing arbitrary predicates are considered, where the semantics of the predicates are specified by underlying theories, allowing to freely combine various theories and to apply theory-specific optimizations within different theory modules. Thus, the core idea to decouple planning specifics from the abstract task corresponds to the work of this thesis, although the motivation differs. Gregory et al. [40] propose to apply the ideas from SMT to planning problems by defining domains in first-order logic with sentences that describe action precondition and effects, analogously to PDDL domain descriptions and to then express extensions to such a problem skeleton by means of theories. The approach was performing competitively on benchmarks versus `Metric-ff`.
In contrast to [40], Bayless et al. [10] propose a monotonic theory as direct addition to the SMT framework that is capable of encoding many interesting problems, such as graph reachability and shortest path problems, which are core problems that are tackled in the field of planning. They compare their solver `MONOSAT` against classical SMT solvers, such as `MINISAT` [34] and `z3` [31] and the Answer Set Programming solver `clasp` [37] with promising results.

Directed Model Checking

A motivation behind directed model checking approaches is to utilize advances in search techniques from the field of automatic planning when verifying or refuting safety properties of systems by trying to find violating paths. Kupferschmid et al. [51] adapt the search heuristics of the prominent *FF Planning System* [44] to refute safety properties, which according to them [52] beats the performance of state-of-the-art model checker tools.

Bogomolov et al. [16] added guided search with the help of *partial plan databases* [5] to the verification of safety properties of hybrid systems [3]. In essence, patterns

correspond to solutions of reachability queries on a subsystem, where subsets of variables and states are ignored. Based on those solutions, heuristics are created by considering minimal distances of all states in the resulting system to the error states of interest. Bogomolov et al. propose to use a simplified version of those pattern on-the-fly, effectively only considering a trajectory to the nearest error state, in order to guide an ongoing reachability into the direction of possible error states. They implemented the approach in the tool `SpaceEx` [36] and evaluated it with the result that their method yields a significant performance boost.

2.6.2 Timed Automata in Planning Domains

In [63] PTAs are used to model the orders of a lacquer production plant. Orders need to pass different stations which can only process one batch at a time. Each order and each station is modeled as a PTA. Stations trigger synchronized actions and ensure mutually exclusive usage. An order schedule was obtained from a reachability analysis using the tool `TaOpt` [64] to the final states of each order in the concurrent system. They compared their results to classical mixed integer linear programming approaches with the conclusion that timed automata can yield competitive results. So in essence their approach modeled existing time constraints, given in the form of orders with deadlines, as timed automata and utilized synchronization features to connect them to the hardware specifics. The reachability analysis then extracts a schedule over the orders.
This is methodically similar to the second approach we explore in Section 4.1 in a way that each constraint is modeled as a separate timed automaton. The difference to our approach is that orders in the lacquer production are independent from each other: While one order impacts another by using a mutually exclusive resource, after processing one order a station can be operated just like before. Therefore the resource usage of one order constraint does not influence the way the same resource is used in other orders. This is not the case in our setting as the different constraints in our approach restrict the pathing of the same platform model automaton concurrently. Therefore if one constraint forces a specific behavior, this has to be recognized by all the other constraint automata as well, which makes the encoding of constraints into timed automata a bit more involved.

Largouët et al. [54] applied PTGAs to planning in multi-agent domains. They use the transport domain from the International Planning Competition (IPC) which is concerned with the delivery of packages using trucks. The vehicles need to manage their fuel and the different distances between package sources and destinations have to be considered. Additionally, they extended the domain by a repair truck and uncontrollable breakdowns of delivery trucks as well as explicit deadlines on package deliveries. The task was solved in two steps by first using the PTA aspects in order to determine the minimal costs and then analyzing the TGA related features to find a strategy that yield the minimal costs while respecting possible breakdowns. They evaluated their results using the existing model checking tools from the `UPPAAL` suite. This work has a similar objective as we pursue as they

generate a plan by means of a reachability analysis but the hurdle that they tackle is the presence of multiple agents that all operate at the same time. In that setting there are multiple automata representing individual entities in the world. Meanwhile in our setting there are only two separate entities, namely the plan and the platform model, that influence each other through defined constraints. On top of that, the plan automaton has a simple structure (it is just a line), because we already have an initial plan that we just have to extend. This makes it easy to explicitly model all possible runs through the composed system and afterwards manipulate that encoding according the given constraints. This is essentially the approach in Section 4.1. Attempting this in a more complex setting like in [54] would be very difficult.

3 Constraint-Based Plan Transformations

With the presented formalisms of Section 2 in mind, let us recap the task that is about to be tackled. In robotic scenarios when hardware-specific details have to be considered while reasoning about high-level objectives, we propose to keep low-level concerns decoupled from abstract domain reasoning and instead model the dependencies separately. Incorporation of platform details into the agent framework is achieved only after an abstract plan has been determined, through a post-processing step that transforms said plan into an executable one. The transformation has to unify the concerns of the different layers, platform specifics on the one side and the abstract plan objective on the other side. This is achieved through the insertions of low-level control actions into the high level plan while respecting temporal constraints expressing the dependencies between the separated entities.

This section is concerned with providing an outline about the capabilities of the plan transformation that is developed throughout this thesis. We first contemplate the requirements such a plan transformation should meet, the objectives it may achieve and the limitations that apply in Section 3.1, before outlining the necessary interfaces between the proposed procedure and high-level reasoning frameworks in Section 3.2. Then the formalisms of Section 2 are brought within the context of our approach, by first discussing the role of timed automata as model for low-level specifics in Section 3.3. Afterwards we revisit *t-ESG* in Section 3.4 in order to express the temporal relations between low-level specifics and high-level plans that build the theoretical foundation of the resulting transformation.

3.1 Requirements and Objectives

A fundamental task, when decoupling platform specifics from the abstract reasoning, is to identify the boundaries of the separation. Specifically, we asked ourselves the following questions:

(I) Which part of the domain problem belong to platform specifications and what forms the abstract domain?

(II) What dependencies connect platform models with the domain?

(III) What are the requirements and restrictions that are imposed on the plan transformation?

Let us preface our take on the above questions by providing examples that we imagine could benefit from a plan transformation based on separately modeled low-level specifics:

As first use-case we consider hardware controls that are not concerned directly with a particular abstract action, but rather depend on the plan structure.

We examine the following scenario of a gripper that is positioned by a system of x, y, z axes, where the precision of the movement degrades with every use. A re-calibration of an axis can be achieved by moving it to one of its endpoints. The need to re-calibrate is determined by the number of abstract `put` or `pick` actions that invoke axis movements. However, re-calibration should only be performed when necessary, depending on the accumulated time spent on the execution of the respective abstract actions.

The specification is not concerned with any particular action, but rather maintains a stable condition of the axis movements throughout the execution by triggering re-calibrations when needed.

As second example we consider the use of low-level actions to refine a complex domain action.

Let us have a look at a scenario where high-level actions depend on fine-grained low-level steps that could be optimized by considering the plan context: Again we base the example on the task to pick up or lay down objects in an abstract domain. A single `pick` operation may depend on several hardware specific steps, e.g., having a running camera to capture data, alignment close to the object based on the provided data, followed by the actual gripping. If the camera should only be running on demand, then the decision to turn off it off may depend on whether `pick` or `put` actions follow in the near future or not.

One could therefore realize the refinement of a `pick` or `put` action via a platform model and formulate constraints to ensure triggering of the necessary low-level calls to physically perform the abstract actions when necessary.

With the above possible use-cases in mind, we state a core assumption towards the role of platform models in the context of abstract domains:

Assumption 3.1.1. *[Towards (I)] A given abstract plan should not be modified, but rather extended, the transformation specifically must not re-order or add/remove domain actions to/from the given plan.*

We believe that low-level specifics should rather refine a given abstract plan, than encode restrictions on the fundamental usability of such a plan. We specifically aim to distinguish the plan transformation from abstract reasoning concerns. As a consequence, this establishes a clear separation of concerns between domain reasoning tasks and the incorporation of platform-specific actions. At the same time boundaries apply to the types of low-level specifics that we can decouple from the high-level domain, namely those that go against Assumption 3.1.1 by requiring re-ordering, adding or removing of domain actions in order to be satisfiable.
The plan transformation also should not invalidate the high-level plan by the

insertion of low-level control actions, which leads us to the assumption stated below:

Assumption 3.1.2 (Towards (II))**.** *Operations of platform components are disjunct from the domain of reasoning.*

By demanding the high-level domain to be completely separated from the scope of the decoupled hardware specifics we can ensure that effects of low-level control actions do not invalidate preconditions of domain actions. In case of an agent given via a *t-ESG* BAT, the contained fluents may not include platform-specific fluents and functions.

Examples for low-level specifications that we deem to be too tight to decouple from the abstract reasoning are given in the following:

Hardware specifics that impose direct restrictions on the actions, a reasoner may select to achieve an objective resemble scenarios that contradict Assumption 3.1.1:

We consider a robot that may pick up objects, but has no capacity to store them before laying them down again. Therefore, in between two `pick` actions there always has to be a `put` action.

Decoupling this specification from the domain interferes with the idea to have a plan transformation separated from the abstract reasoning system. We argue that a transformation capable of transforming a plan by significantly altering the abstract plan semantics raises the question whether calculating such an abstract plan is beneficial in the first place when a considerable amount of reasoning is applied later. While such a strong transformation may make sense in some applications, it does collide with the idea to control platform models without deeper knowledge of the reasoning backend.

Possible problems that Assumption 3.1.2 prevents, arise when low-level operations that modify the domain of reasoning:

As an example, let there be a mobile robot which has to get its batteries exchanged from time to time. In order to do so, it has to drive to a docking station. Modeling this via a specific component can be reasonable, e.g., if there are also robots working in the same domain that have sufficient power supply for the whole operation span. Then a domain without battery-specific information may be used on both kind of robots, while a plan transformation adapts the plans on those robots that have to recharge.

The assertion of actions during plan transformations that alter the abstract domain introduces the following problem: There is no guarantee that the resulting plan is executable anymore by the means of the domain specification. In the above example a platform designer might be tempted to create a model that forces the insertion of a `go-charge` action, when the battery status reaches a certain threshold,

in order to drive to a docking station. A simple plan, such as

$$\texttt{goto}(\text{START}, \text{TABLE}), \texttt{goto}(\text{TABLE}, \text{DOOR})$$

may be transformed into

$$\texttt{goto}(\text{START}, \text{TABLE}), \texttt{go-charge}(\text{POWER-DOCK}), \texttt{goto}(\text{TABLE}, \text{DOOR})$$

effectively violating a possible precondition of `goto`, that the source of movement must coincide with the current position.

The battery recharging example was chosen as it resembles a problem instance that is rather trivial to properly handle: A platform that demands to drive to a power station could simply move to the original position again afterwards or even better, the plan transformation could recognize the need to adapt the source location of the `goto` action. However, the generalization to detect and properly model the allowed interferences with a high-level domain would require to make assumptions about it, which we like to abstain from, not only due to attempted separation of concerns, but also to keep the proposed plan transformation as compatible as possible with the different types of reasoners. By basing constraints on the plan structure without caring about the induced implications on the high-level domain, the only requirement to use the transformation is to actually produce plans. This also means that a properly modeled component to manage the battery power can be used within our approach, as there is no notion to prevent a usage that simply inserts platform actions in a invariant way that does not harm the high-level reasoning. However, we emphasize that this contradicts the proposed separation of concerns as a platform designer would have to equate for the implications to the feasibility of domain plans.

We believe that in its core a separation of low-level specifics from the abstract domain should enable the enforcement of precise temporal control over the hardware. Independent of whether the abstract domain is concerned with temporal constraints or not, in lower layers of a robotic platform timings often play a crucial role. As evidence, we consider sensors that have to run for a while to produce noise-corrected data, components that take time boot up and shut down or different hardware that interferes with each other, when running at overlapping times, e.g., depth cameras producing noise in infrared sensor data. Therefore, we propose the following:

Assumption 3.1.3 (Towards (II))**.** *Dependencies between platform models and abstract plans are modeled via temporal constraints that describe temporal control of platform actions based on action patterns occurring in the plans.*

While temporal aspects may be a important both during high-level reasoning to provide an abstract plan and during the post-processing step of applying a transformation in order to meet low-level specific requirements, we argue that the responsibility to control the overall execution has to be on the side of the reasoning framework.

With all the preceding assumptions in mind, we determine the role of the plan transformation in a robotic framework as follows:

Assumption 3.1.4. *[Towards (III)] The reasoning framework produces sequential plans together with temporal constraints that form a sufficient condition on the high-level feasibility of said plans. Based on*

- *the temporal restrictions propagated by the reasoner,*
- *the control restrictions induced by the models and*
- *the temporal constraints expressing relations between high-level and low-level*

the plan transformation determines an executable plan together with execution start times for each action.

Note how the above assumption goes in accordance to Assumption 3.1.1 by that the reasoning of the high-level framework is not altered, but rather interpreted as hard constraint. Therefore the different concerns of domain reasoning and platform specific adaptions are clearly distinguished.

3.2 Overview of the Procedure

Under consideration of the assumptions proposed in Section 3.1, we can now describe the steps required to form a plan transformation procedure that unifies abstract plans acquired through high-level reasoning with separately modeled platform-specific low-level requirements. Figure 3.1 shows schematics of the full procedure that we outline in the following:
A platform designer models lower level specifics as timed automata, explained further in Section 3.3, and defines their control-based on the domain context via constraints in a subset of the logic $t\text{-}\mathcal{ESG}$. The constraint formulas describe the behavior of the timed automata models through specifying domain action patterns and their effect on the automata behavior. Details on the constraint formulas are given in Section 3.4. Timed automata, constraints and the abstract plan are then encoded to form a reachability model checking problem, such that there exists a transformed plan, if and only if there is a solution path for that encoded problem. Approaches for the encoding step are covered later in Section 4. Solution paths are returned in the form of symbolic traces and a decoding step is necessary in order to obtain a transformed plan together with execution times for the fastest possible execution. The resulting executable plan can then be handed back to the execution framework. In case practical issues cause the given execution times to be missed, the decoder may be able to give updated execution times that still fulfill all constraints by calculating a different solution from the symbolic trace that incorporates the occurred delays. The decoding steps are detailed in Section 5.

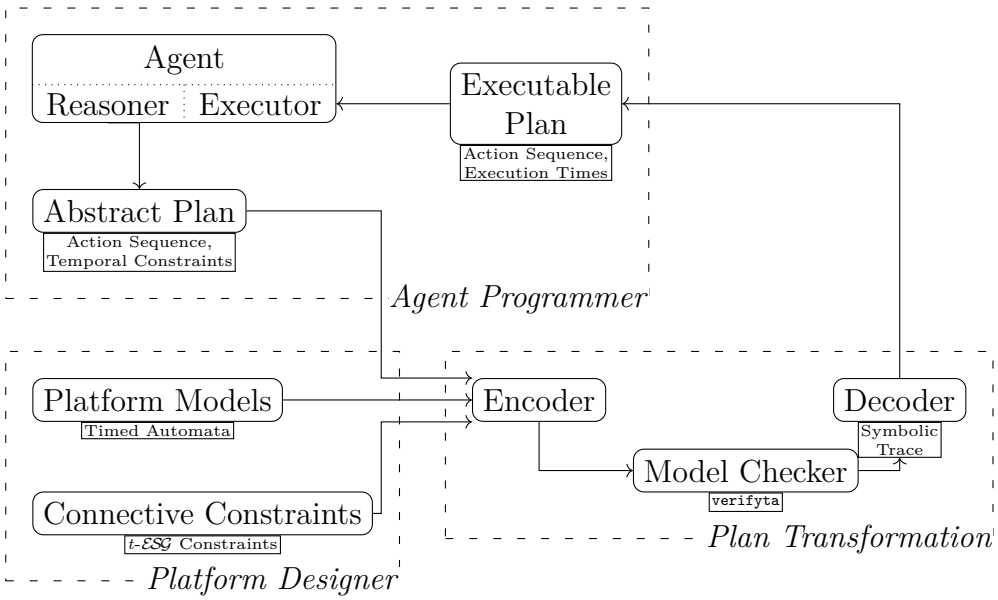


Figure 3.1: Steps to transform abstract plans into executable ones based on low-level models.

3.3 Platform models as Timed Automata

We propose to model platform specifics as timed automata such that the platform functions are represented by locations. This is due to the fact that time only elapses within locations and taking transitions consumes no time. Functionalities of a component $\mathcal{M}$ are triggered by the agent via actions that form the alphabet Σ of the corresponding automaton $\mathcal{A}_{\mathcal{M}}$. Therefore, the need to trigger a low-level action corresponds to a state change within the automaton model. In case of the automaton in Figure 2.1 the alphabet is $\Sigma = \{\texttt{no-op}, \texttt{turn-on}, \texttt{shut-off}\}$.

We also assume different platform models to be independent from each other in a sense that there are no temporal dependencies between components and that actions of different platforms can be executed in parallel. We argue that in case different models influence each other, they could be merged together such that their relations are encoded within the timed automaton through clock constraints. This restriction allows us to develop a plan transformation procedure with the only concern being to satisfy connective constraints between platform models and domain actions, without having to consider implications that dependencies between different platform models may entail.

Lastly, we re-iterate that Assumption 3.1.4 implies that it is out of the scope of the proposed plan transformation to prevent or even just detect models that potentially mix low-level concerns with the abstract domain against Assumption 3.1.1, because it is designed to only deal with the resolution of temporal constraints. The responsible is therefore on the side of platform model designers to not construct low-level interfaces that temper with the feasibility of high-level plans.

3.4 Constraints

Dependencies between the abstract agent and platform models may include qualitative time constraints, as well as quantitative constraints, e.g., to specify time windows during which a component needs to be operated in order to proceed with the plan.
We aimed to obtain constraints that are capable of expressing the basic qualitative relations between platform operations and domain actions, similar to Allen's interval algebra [1], with the possibility to quantify the interval lengths. As an overview, the basic relations between intervals are depicted in Figure 3.2. However, in the context of plan transformations where a pattern that requires the insertion of low-level actions may occur multiple times throughout a plan, the classical notion of interval relations does not really fit our needs. In particular, notions like *component $\mathcal{M}$ should be in state* $\texttt{loc}_1$ *strictly after the execution of* $\texttt{action}_1()$ do not make sense in our application, where visiting $\texttt{loc}_1$ after one instance of $\texttt{action}_1()$ may also mean it is visited before the next instance of $\texttt{action}_1()$. We rather take the classical interval relations as guideline and provide constraints that locally fulfill such relations ignoring their global context.

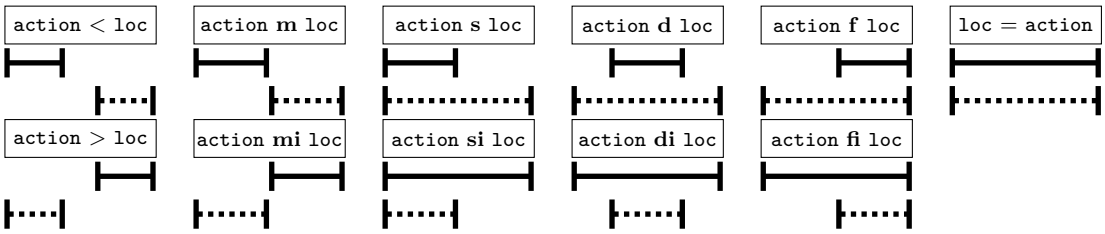


Figure 3.2: Relations describing local relations between platform locations (dotted intervals) and high-level action durations (solid intervals).

We consider two different types of constraints to establish connections between high-level action patterns and platform model operations. Firstly, constraints that are concerned with operations near the execution of a plan action, such as

when $\texttt{action}_i$ **occurs**, **do** `platform_operations` **afterwards/beforehand/now**.

Secondly, platform operations in between the execution of high-level actions, e.g.,

when $\texttt{action}_i$ **occurs**, **later followed by** $\texttt{action}_j$ **do** `platform_operations` **in between**.

Together they cover our needs with respect to the relations in Figure 2.6, as we demonstrate later.

3.4.1 Constraints Based on the Occurrence of Actions

Let us begin with the former constraints and derive formulas that the plan transformation should be capable of handling. The resulting formulas have the form:

$$[\![\delta]\!]\mathbf{G}\left[\Psi_P \supset \Psi_{\mathcal{M}}\right],$$

where Ψ_P describes a situation during the execution of δ which requires to operate the platform $\mathcal{M}$ and $\Psi_{\mathcal{M}}$ specifies the usage of $\mathcal{M}$ in that situation. Given Ψ_P and Ψ_M we also write $\mathfrak{occ}(\Psi_P, \Psi_M)$ to denote such constraints. The reasoning behind the proposed structure given above is presented now.

Semantics of $[\![\delta]\!]\phi$ in the Context of Plan Transformations $\mathfrak{occ}$ formulas utilize the capability of *t-ESG* to restrict traces belonging to the execution of programs. Specifically, we take a closer look at the semantics of situation formulas according to Definition 2.4.8, that have the form $[\![\delta]\!]\phi$, with ϕ being a trace formula.

In *t-ESG* δ refers to a program description via Definition 2.4.3, however, the semantics of $[\![\delta]\!]\phi$ are only concerned with traces of said program, which correspond to the possible action sequences produced by δ. In our context we are confronted with one given plan at a time, that has to be transformed. So for us, δ can be interpreted as a sequential plan, rather than a program description.

We illustrate how the *t-ESG* semantics enable the formalization of plan transformations by means of the following example:.

Let us consider a formula stating that five to seven seconds after each execution start of an action $\texttt{action}_1()$ the state $\texttt{loc}_1$ of platform `Model` should be reached:

$$\gamma := [\![\delta]\!]\mathbf{G}\left[Occ(\texttt{action}_1()) \supset \mathbf{F}_{[5,7]}\mathit{state}(\texttt{Model}) = \texttt{loc}_1\right]$$

Also, let $p = \langle(\texttt{action}_1(), t_1), (\texttt{action}_2(), t_1 + 15)\rangle$ be the abstract plan to transform, thus stating that $\texttt{action}_2$ follows 15 seconds after $\texttt{action}_1$. γ essentially implies the structure of possible transformed plans for any high-level plan. For the given plan p we are confronted with the task to find a world with empty history $\langle\rangle$ (assuming an initial world state), such that γ holds during the execution of the transformation $\hat{p}$ of p. In a first step, we remove the short-hand notations such that we can apply the semantics of *t-ESG* formulas (Definition 2.4.8 and 2.4.9):

$$\begin{aligned}\gamma :=& [\![\delta]\!]\mathbf{G}\left[Occ(\texttt{action}_1()) \supset \mathbf{F}_{[5,7]}\mathit{state}(\texttt{Model}) = \texttt{loc}_1\right]\\ =& [\![\delta]\!]\neg\left[\mathbf{F}\neg\left[Occ(\texttt{action}_1()) \supset \mathbf{F}_{[5,7]}\mathit{state}(\texttt{Model}) = \texttt{loc}_1\right]\right]\\ =& [\![\delta]\!]\neg\left[\top\mathbf{U}\neg\left[Occ(\texttt{action}_1()) \supset \top\mathbf{U}_{[5,7]}\mathit{state}(\texttt{Model}) = \texttt{loc}_1\right]\right]\end{aligned}$$

Now the definitions from Section 2.4 can be applied in order to satisfy γ for a world w with history $\langle\rangle$:

$$
\begin{aligned}
w, \langle\rangle \models & [\![\delta]\!] \neg \Big[\top \mathbf{U} \neg \Big[Occ(\texttt{action}_1()) \supset \top \mathbf{U}_{[5,7]} state(\texttt{Model}) = \texttt{loc}_1 \Big]\Big] \\
\text{iff} \quad & w, \langle\rangle, \hat{p} \models \neg \Big[\top \mathbf{U} \neg \Big[Occ(\texttt{action}_1()) \supset \top \mathbf{U}_{[5,7]} state(\texttt{Model}) = \texttt{loc}_1 \Big]\Big] \\
\text{iff} \quad & w, \langle\rangle, \hat{p} \not\models \Big[\top \mathbf{U} \neg \Big[Occ(\texttt{action}_1()) \supset \top \mathbf{U}_{[5,7]} state(\texttt{Model}) = \texttt{loc}_1 \Big]\Big] \\
\text{iff} \quad & \text{for all } z_1 \text{ s.t. } \hat{p} = z_1 \cdot p' \text{ the following holds:}
\end{aligned}
$$

$$
\begin{aligned}
w, z_1, p' \not\models & \neg \Big[Occ(\texttt{action}_1()) \supset \top \mathbf{U}_{[5,7]} state(\texttt{Model}) = \texttt{loc}_1 \Big] \\
\text{iff} \quad & w, z_1, p' \models \Big[Occ(\texttt{action}_1()) \supset \top \mathbf{U}_{[5,7]} state(\texttt{Model}) = \texttt{loc}_1 \Big] \\
\text{iff} \quad & w, z_1, p' \not\models Occ(\texttt{action}_1()) \text{ or } w, z_1, p' \models \top \mathbf{U}_{[5,7]} state(\texttt{Model}) = \texttt{loc}_1 \\
\text{iff} \quad & w, z_1, p' \not\models Occ(\texttt{action}_1()) \text{ or exists } z_2 \text{ s.t. } p' = z_2 \cdot \hat{p} \text{ and:}
\end{aligned}
$$

$w, z_1 \cdot z_2, \hat{p} \models state(\texttt{Model}) = \texttt{loc}_1$ and $time(z_1 \cdot z_2) \in time(z_1) + [5, 7]$

This provides us with the necessary information to understand how γ demands adaptions of p leading to a transformed plan $\hat{p}$. Given that we have three candidates for z_1 in p, namely

(1) $\langle\rangle$,

(2) $\langle(\texttt{action}_1, t_1)\rangle$ and

(3) $p = \langle(\texttt{action}_1, t_1), (\texttt{action}_2, t_1 + 15)\rangle$,

we observe that for (1) and (3) $w, z_1, p' \not\models Occ(\texttt{action}_1())$ holds, therefore only case (2) is left to consider. However, there is no such z_2 that could satisfy

$$
\begin{aligned}
& time(z_1 \cdot z_2) \in time(z_1) + [5, 7] \\
\Leftrightarrow & time(z_1 \cdot z_2) \in [t_1 + 5, t_1 + 7]
\end{aligned}
$$

given the possible candidates:

(1) $z_2 = \langle\rangle$, then $time(z_1 \cdot z_2) = time(z_1) = t_1 \notin [t_1 + 5, t_1 + 7]$ and

(2) $z_2 = \langle(\texttt{action}_2(), t_1 + 15)\rangle$ with $time(z_1 \cdot z_2) = t_1 + 15 \notin [t_1 + 5, t_1 + 7]$.

Therefore, p must be extended by some entries to satisfy γ.

Let us first consider a scenario, where `Model` is already in $\texttt{loc}_1$ before the execution of p and does not change its state at any time. Then p does not satisfy γ due to the fact that there is no time point $[t_1 + 5, t_1 + 7]$ covered in p, even tho $state(\texttt{Model}) = \texttt{loc}_1$ holds there. In fact, the point-wise semantics of *t-ESG* trace formulas require to additionally insert a dummy action into p in order to observe that the required state actually is reached within the given time frame. This can be achieved by a designated `observe` action that has no preconditions and no effects. `observe(Model)` merely equates to explicitly checking a platform models' current state and has no relevance for execution. Since the proposed plan transformation is aimed towards usage in real-world scenarios, where the accuracy of time steps is practically determined, e.g, through the clock frequencies of the used cpu cores, we may safely assume the observable time to be discretized through some minimal time delta. Hence we may assume that each transformed plan contains `observe`

actions at each discrete step, which effectively corresponds to the (practically) continuous monitoring of low-level components during execution.

The introduction of an `observe` action solves the case of p' requiring no further platform interactions as the target state $\texttt{loc}_1$ was already reached.
If we instead assume `Model` to be in $\texttt{loc}_0$ when p starts, then γ requires the insertion of appropriate control actions to ensure a state change to $\texttt{loc}_1$ at the specified time frame. One possible transformed plan satisfying γ, assuming `Model` may change from $\texttt{loc}_0$ to $\texttt{loc}_1$ at any time via some action $a_{\texttt{Model}}$ and ignoring unnecessary `observe` actions, can be given as:

$$\hat{p} = \langle(\texttt{action}_1, 0), (a_{\texttt{Model}}, 4), (\texttt{observe}(\texttt{Model}), 5), (\texttt{action}_2, 15)\rangle$$

Finally, we highlight the role of the operator $\mathbf{G}$ in the proposed type of constraints: It enforces platform operations on every occurrence of a pattern Ψ_P, allowing us to essentially state invariant conditions that maintain the executability of high-level actions throughout the scope of a plan.

Specification of Action Patterns in Ψ_P *t-ESG* allows complex descriptions of action patterns within traces, having the potential to significantly complicate the transformation procedure. It might be tempting to define dependencies between high-level behavior and low-level specific by associating certain domain predicates with the need to operate a platform. Let us consider the following example to demonstrate problems that can arise:

A robot doing grasping tasks has a domain predicate *Holding* and a platform component $\mathcal{M}_{\texttt{grip}}$ taking care of gripper movements. Domain actions include several different operations to pick up objects, such as `get_from_ground` and `get_from_conveyor_belt`, the hardware demands the gripper to move to a position `parked` when objects are held to reduce stress on the joints. A constraint such as

$$\gamma_{\texttt{grip}} = [\![\delta]\!]\mathbf{G}\exists x.\mathit{Holding}(x) \supset \mathbf{F}_{[0,5]}\mathit{state}(\mathcal{M}_{\texttt{grip}}) = \texttt{parked}$$

would be very handy to state that the gripper gets parked independently which action causes the predicate *Holding* to be true. However, this can be problematic, depending on the successor state axiom of *Holding*. Consider the following axiom stating that `get_from_conveyor_belt` only grips something every other ten seconds:

$$\begin{aligned}\Box[a]\mathit{Holding}(x) \equiv\ & a = \texttt{get\_from\_ground}(x)\\ & \vee (a = \texttt{get\_from\_conveyor\_belt}(x) \wedge \mathit{time}(a) \mod 20 < 10)\\ & \vee (\mathit{Holding}(x) \wedge a \neq \texttt{drop}(x))\end{aligned}$$

Then the activation of $\gamma_{\texttt{grip}}$ depends on the action timings of the abstract plan actions. If we assume a trace $\langle(\texttt{get\_from\_conveyor\_belt}, t_1)\rangle$ with $t_1 < 20$ then it is unclear whether $\gamma_{\texttt{grip}}$ should enforce the gripper to move in the parking

position or not, because the exact action timing is only determined through the transformation itself. The problem is circumvented if we formulate $\gamma_{\texttt{grip}}$ based on the occurring predicate instead, via

$$[\![\delta]\!]\mathbf{G}\exists x.\mathit{Occ}(\texttt{get\_from\_conveyor\_belt}(x)) \vee \mathit{Occ}(\texttt{get\_from\_ground}(x)) \supset \mathbf{F}_{[0,5]}\mathit{state}(\mathcal{M}_{\texttt{grip}}) = \texttt{parked}.$$

Although it may same situation, the subtle difference is that the plan formation does not know anything about the actual effects that domain actions carry out. Due to Assumption 3.1.4 it is merely responsible to respect the temporal relations between different actions, without further knowledge about the carried semantics. Hence, if the high-level reasoner allows the execution of grasping times during intervals, where no object can be grabbed from the conveyor, then we can assume that the action is executable, hence it may be started. We argue that it is not the responsibility of a plan transformation to also acknowledge, whether the execution if domain actions yields the intended results.

For simplicity we only allow constraints, where it can be determined in advance which patterns in a given abstract plan enforces behavior of the platform. However, at the same time we do not want to impose restrictions on the successor state axioms of the domain for two reasons: On the one hand we deem it undesirable if the expressive power of platform constraints restricts the abstract domain and thereby forcing trade-offs between the reasoning capabilities of the high-level system and the benefits of context dependent hardware control. On the other hand many reasoning systems do not have explicit successor state representation of predicates, because they are not based on the situation calculus. To keep the ideas proposed in this thesis compatible with arbitrary plan-based reasoners we instead formulate Ψ_P solely based on the structure of the plan. This can be achieved within the *t-ESG* formalism by introducing a designated predicate $\mathit{Occ}(x)$ with successor state axiom

$$\Box[a]\mathit{Occ}(x) \equiv a = x$$

as it was already presented in 2.4.

Specification of Platform Behavior in $\Psi_{\mathcal{M}}$ It remains to formalize the temporal relations between platform behavior and the occurrence of domain actions. Precice timed control can be ensured by utilizing the trace operators $\mathbf{U}_I$ and $\mathbf{S}_I$ (and the induced short-hand notations $\mathbf{G}_I$,$\mathbf{H}_I$, $\mathbf{F}_I$ and $\mathbf{P}_I$) with attached intervals I. The operators $\mathbf{X}_I$ and $\mathbf{V}_I$ are not suited to express plan transformation rules, because of the previously noted possibility of elements being inserted into a trace. In particular, a constraint of the form

$$\gamma_X = [\![\delta]\!]\mathbf{G}\,\mathit{Occ}(\texttt{action}_1()) \supset \mathbf{X}_{[8,9]}\mathit{state}(\texttt{Model}_2) = \texttt{loc}_2$$

would interact with the constraint γ from the above example, even tho they argue about different platform models, which we assumed to be fully independent from each other: The trace $\delta_{\texttt{res}}$ cannot possibly be extended to also satisfy γ_X as it

would require the insertion of $\texttt{observe}(\texttt{Model}_2)$ directly after entry of $\texttt{action}_1()$, but at the same time $\texttt{observe}(\texttt{Model}, 5)$ has to be in between $\texttt{observe}(\texttt{Model}_2)$ and $\texttt{action}_1()$.

For $\Psi_{\mathcal{M}}$ we describe the desired behavior of $\mathcal{M}$ using a special fluent *state* that takes as input a component name and returns the current state of it. Trace operators may be used to enforce future or past behavior of the component within time bound I.

Now we are ready to generalize constraints through the grammar in Figure 3.3. We note that negations of $\langle actionFormula \rangle$ formulas is prohibited due to the fact that this could cause infinity loops during transformation, e.g., a constraint stating that whenever $\neg Occ(\texttt{action}_1())$ holds, some platform has to be operated, causes the insertion of low-level actions as entries in the plan trace, for those entries $\neg Occ(\texttt{action}_1())$ holds again.

$$
\begin{aligned}
\langle constraint \rangle &::= [\![\delta]\!]\mathbf{G}\,[\langle actionFormula \rangle \supset \langle stateFormula \rangle] \\
\langle actionFormula \rangle &::= \mid Occ(\texttt{action}(\vec{x})) \mid \langle actionFormula \rangle \vee \langle actionFormula \rangle \\
\langle traceOpUnary \rangle &::= \mathbf{F}_I \mid \mathbf{P}_I \mid \mathbf{G}_I \mid \mathbf{H}_I \\
\langle traceOpBinary \rangle &::= \mathbf{S}_I \mid \mathbf{U}_I \\
\langle stateFormula \rangle &::= \langle atomicSF \rangle \\
&\quad \mid \langle traceOpUnary \rangle \langle atomicSF \rangle \\
&\quad \langle atomicSF \rangle \langle traceOpBinary \rangle \langle atomicSF \rangle \\
\langle atomicSF \rangle &::= state(\texttt{Model}) = \texttt{loc} \\
&\quad \mid \langle atomicSF \rangle \vee \langle atomicSF \rangle \\
&\quad \mid \neg \langle atomicSF \rangle
\end{aligned}
$$

Figure 3.3: BNF for constraints expressing the relations between occurring actions and platform operations. `action` refers to an action standard name, $\vec{x}$ denotes action parameters (either free variables that are implicitly existentially quantified or standard names) and `loc` denotes a name identifier of a location from a component `Model`

3.4.2 Constraints Based on the Duration of Actions

The previously presented constraints lack an important feature: They cannot explicitly guide platform operations based on the duration of high-level patterns, because they only state behavior based on the occurrence of domain actions. To overcome those issues we also provide another type of constraints that allow a precise description of platform operations bounded in between the occurrences of

two domain actions: Given a finite set $B = (\alpha_j, I_j)_{1 \leq j \leq n}, n \in \mathbb{N}$ with α_j denoting formulas built through the grammar of $\langle atomicSF \rangle$ of Figure 3.3, I_j denoting intervals according to the *t-ESG* syntax, and given β_1, β_2 formulas built through the grammar of $\langle actionFormula \rangle$ stated in Figure 3.3, we define *until-chain constraints* as:

$$\begin{aligned}\mathfrak{uc}(B, \beta_1, \beta_2) := [\![\delta]\!]\mathbf{G}(\beta_1 \wedge \mathbf{X}\neg\beta_1\mathbf{U}\beta_2) \supset \\ (\alpha_1 \wedge \neg\beta_2)\mathbf{U}^s_{I_1}(\alpha_2 \wedge \neg\beta_2)\mathbf{U}^s_{I_2} \ldots (\alpha_n \wedge \neg\beta_2)\mathbf{U}^s_{I_n}\beta_2,\end{aligned}$$

where $\phi_1\mathbf{U}^s_I\phi_2 := \phi_1 \wedge (\phi_1\mathbf{U}\phi_2)$.

An until-chain constraint essentially states, that whenever an action pattern β_1 is followed without further occurrences of β_1 by another action pattern β_2, then the platform operations between such a matching pair of domain actions a_1 and a_2 are given by sequentially visiting the states specified in B. Specifically, B defines the platform usage by providing a sequence of target states $L^i_{\texttt{target}}$ through the formulas α_i, such that the component separately stays within each $L^i_{\texttt{target}}$ for a duration within I_i. Even if $L^{i+1}_{\texttt{target}} = L^i_{\texttt{target}}$, then $L^i_{\texttt{target}}$ is visited for some time within I_i, before the component remains in $L^{i+1}_{\texttt{target}}$ for another duration within I_{i+1}. The enforcement of states from $L^1_{\texttt{target}}$ starts with the execution start of a_1, the requirement to be in $L^n_{\texttt{target}}$ ends with the execution start of a_2.

Let us consider the case of $n = 1$, with a constraint

$$\gamma_\Delta := \mathfrak{uc}\left[\langle(state(\mathcal{M}) = \texttt{running}, [0, \infty))\rangle, Occ(\texttt{start\_pick}(o)), Occ(\texttt{end\_pick}(o))\right],$$

stating that during each execution of pick the component $\mathcal{M}$ has to be in state `running`. We break down the constraint in order to explain the purpose of its' different parts:

$$\begin{aligned}\gamma_\Delta = [\![\delta]\!]\mathbf{G}\Big[&\underbrace{Occ(\texttt{start\_pick}(o))}_{(I)} \wedge \underbrace{\mathbf{X}(\neg Occ(\texttt{start\_pick}(o))\mathbf{U}\, Occ(\texttt{end\_pick}(o)))}_{(II)} \\ &\supset (\underbrace{state(\mathcal{M}) = \texttt{running}}_{(III)} \wedge \underbrace{\neg Occ(\texttt{end\_pick}(o))}_{(IV)})\, \underbrace{\mathbf{U}^s_{[0,\infty)}}_{(V)}\, \underbrace{Occ(\texttt{end\_pick}(o))}_{(VI)}\Big]\end{aligned}$$

Similar to $\mathfrak{occ}$ constraints, (I) essentially describes the high-level action pattern, here the occurrence of a `start_pick` action, that requires platform control. Additionally, (II) ensures that

- `start_pick` is actually followed by a matching `end_pick` later and
- between the current instance of `start_pick` and the next matching `end_pick`, there is no other `start_pick` action.

Upon finding a suitable candidate at (I), the platform has to be in state `running`, specified at (III), until the matched occurrence according to (II) is encountered in (VI), namely the end of the started pick action. The additional requirement of (IV) scopes the matching `end_pick` to the next possible occurrence, instead of an arbitrary one from the remainder of the plan. Generally, (II) and (IV)

together imply that only the smallest matchings across different instances of β_1 and β_2 are considered to be relevant for the platform behavior specified through B. Lastly, the modified until operator $\mathbf{U}^s$ is used in (V) to guarantee that the state `running` is actually observed. If the normal operator $\mathbf{U}$ was used, then a plan $\langle(\texttt{start\_pick}(o), 0), (\texttt{end\_pick}(o), 15)\rangle$ would already satisfy γ_Δ per the given semantics of $\mathbf{U}$ in Definition 2.4.9.

A different example for the utility provided by $\mathfrak{uc}$ formulas is the following constraint $\gamma_{\texttt{precise}}$ that contains a sequence B of length 3:

$$\gamma_{\texttt{precise}} := \mathfrak{uc}\Big[\langle(\top_{\mathcal{A}}, I_\top), (\mathit{state}(\mathcal{M}) = \texttt{calibrated}, I_{[5,5]}), (\mathit{state}(\mathcal{M}) = \texttt{parked}, I_\top)\rangle, \\ \mathit{Occ}(\texttt{fast\_pick}(o)), \mathit{Occ}(\texttt{precise\_pick}(o'))\Big]$$

where $\top_{\mathcal{A}}$ denotes a disjunction over all states of $\mathcal{A}$ and $I_\top$ denotes the unbounded interval $[0, \infty)$. $\gamma_{\texttt{precise}}$ could model the need to calibrate a gripper whenever a `fast_pick` action precedes a `precise_pick` action. After calibration the gripper has to remain in a parked state such that it does not lose the calibration due to possible collisions with other objects. The sequence in B allows unconstrained control of $\mathcal{M}$ when a `fast-pick` action starts, until eventually the calibrated state is visited for precisely 5 seconds followed by a remain in `parked` until `precise_pick` starts.

Until-chain constraints together with the normal occurrence-based constraints are capable to express relations similar to the ones in Figure 3.2 as we summarize in Table 3.1.

3.4.3 Constraints from the High-Level Domain

Now we are able to express temporal relations between plan actions and platform states via the constraints above and between different platform states by directly encoding them in the timed automaton. Temporal relations between different plan actions are not covered yet. Complex constraints affecting the ordering of actions have to be dealt with during the actual abstract planning already and therefore should not concern the transformation into an executable plan.

We deem the relevant information to be covered through constraints about the start of each action in a plan. To that end we provide two different types of formulas acting as interface to translate the temporal restrictions of the execution to the scope of the transformation. On the one hand the reasoner may require each action in a plan to be started within a certain time interval. As a prerequisite we define a function *PlanOrder* that allows to address the n-th plan action independent from the performed steps of the plan transformation. Essentially, *PlanOrder* has to count the number of number of domain actions that have been started already. The successor state axiom of *PlanOrder* is shown below, it utilizes a designated predicate *IsDomAct* that is true precisely for every plan action.

$\Gamma_{=} = \{\mathfrak{uc}\,[\langle(\alpha_{\texttt{loc}}, I_\top)\rangle, \beta_{\texttt{start}}, \beta_{\texttt{end}}]\}$	During `action` the state is `loc`.
$\Gamma_{<} = \{\mathfrak{uc}\,[B_{\texttt{after}}, \beta_{\texttt{end}}, \beta_{\texttt{start}}]\,,$ $\mathfrak{uc}\,[B_{\texttt{after}}, \beta_{\texttt{end}}, \beta_{\texttt{last}}]\}$	after each `action` occurrence, `loc` should be avoided for $[v, w]$ seconds, then `loc` should be visited for at least $[x, y]$ seconds afterwards.
$\Gamma_{>} = \{\mathfrak{uc}\,[B_{\texttt{before}}, \beta_{\texttt{end}}, \beta_{\texttt{start}}]\,,$ $\mathfrak{uc}\,[B_{\texttt{before}}, \beta_{\texttt{first}}, \beta_{\texttt{start}}]\}$	Before each `action` occurrence, `loc` should be avoided for $[v, w]$ seconds after `loc` was visited for at least $[x, y]$ seconds.
$\Gamma_{\mathbf{m}} = \{\mathfrak{occ}\left[\beta_{\mathbf{end}}, \alpha_{\texttt{loc}}\mathbf{U}_{[x,y]}\bar{\alpha}_{\texttt{loc}}\right],$ $\mathfrak{uc}\,[\langle(\bar{\alpha}_{\texttt{loc}}, I_\top)\rangle, \beta_{\texttt{start}}, \beta_{\texttt{end}}]\}$	During `action`, `loc` should be avoided, right after `action` ends, `loc` should be visited for at least $[x, y]$ seconds.
$\Gamma_{\mathbf{mi}} = \{\mathfrak{occ}\left[\beta_{\mathbf{start}}, \alpha_{\texttt{loc}}\mathbf{S}_{[x,y]}\bar{\alpha}_{\texttt{loc}}\right],$ $\mathfrak{uc}\,[\langle(\bar{\alpha}_{\texttt{loc}}, I_\top)\rangle, \beta_{\texttt{start}}, \beta_{\texttt{end}}]\}$	During `action`, `loc` should be avoided, right before `action` starts, `loc` should be visited for at least $[x, y]$ seconds.
$\Gamma_{\mathbf{s}} = \Gamma_{=} \cup \{\mathfrak{occ}\left[\beta_{\mathbf{end}}, \alpha_{\texttt{loc}}\mathbf{U}_{[x,y]}\bar{\alpha}_{\texttt{loc}}\right]\}$	During `action` the state is `loc`. Afterwards, `loc` is continued to be visited for $[x, y]$ seconds.
$\Gamma_{\mathbf{si}} = \{\mathfrak{uc}\Big[\langle(\alpha_{\texttt{loc}}, [x, y]), (\bar{\alpha}_{\texttt{loc}}, I_\top)\rangle,$ $\beta_{\texttt{start}}, \beta_{\texttt{end}}\Big]\}$	When `action` starts, `loc` is visited for $[x, y]$ seconds and then avoided until `action` ends.
$\Gamma_{\mathbf{f}} = \Gamma_{=} \cup \{\mathfrak{occ}\left[\beta_{\mathbf{start}}, \alpha_{\texttt{loc}}\mathbf{S}_{[x,y]}\bar{\alpha}_{\texttt{loc}}\right]\}$	During `action` the state is `loc`. Before that, `loc` was already visited for $[x, y]$ seconds.
$\Gamma_{\mathbf{fi}} = \{\mathfrak{uc}\Big[\langle(\bar{\alpha}_{\texttt{loc}}, I_\top), (\alpha_{\texttt{loc}}, [v, w])\rangle,$ $\beta_{\texttt{start}}, \beta_{\texttt{end}}\Big]\}$	During `action` `loc` is only visited in the last $[v, w]$ seconds.
$\Gamma_{\mathbf{d}} = \Gamma_{\mathbf{f}} \cup \Gamma_{\mathbf{s}}$	During `action` the state is `loc`. Also visited directly before `action` `loc` is visited for $[v, w]$ seconds. Also, directly after `action` `loc` is visited for $[x, y]$ seconds.
$\Gamma_{\mathbf{di}} = \{\mathfrak{uc}\Big[\langle(\bar{\alpha}_{\texttt{loc}}, [v, w]), (\alpha_{\texttt{loc}}, [x, y]),$ $(\bar{\alpha}_{\texttt{loc}}, I_\top)\rangle, \beta_{\texttt{start}}, \beta_{\texttt{end}}\Big]\}$	When `action` starts, `loc` is avoided for $[v, w]$ seconds and then visited for $[x, y]$ seconds before `action` ends.

Shorthand Notations

$\alpha_{\texttt{loc}} := \mathit{state}(\texttt{Model}) = \texttt{loc}$	$\beta_{\texttt{start}} := \mathit{Occ}(\texttt{start\_action}())$	$\beta_{\texttt{first}} := \mathit{Occ}(\texttt{first}())$
$\bar{\alpha}_{\texttt{loc}} := \neg\alpha_{\texttt{loc}}$	$\beta_{\texttt{end}} := \mathit{Occ}(\texttt{end\_action}())$	$\beta_{\texttt{last}} := \mathit{Occ}(\texttt{last}())$
$B_{\texttt{after}} := \langle(\bar{\alpha}_{\texttt{loc}}, [v, w]),$ $(\alpha_{\texttt{loc}}, [x, y]),$ $(\top, I_\top)\rangle$	$B_{\texttt{before}} := \langle(\top, I_\top,$ $(\alpha_{\texttt{loc}}, [v, w]),$ $(\bar{\alpha}_{\texttt{loc}}, [x, y])\rangle$	$I_\top := [0, \infty)$

Table 3.1: Modeling the basic interval relations as depicted in Figure 3.2, assuming dedicated `first` and `last` actions to denote the begin and end of a plan.

$$\begin{aligned}\Box[a]\mathit{PlanOrder} = n \equiv &(n = 0 \wedge \neg \mathit{IsDomAct}(a) \wedge \mathbf{H}\forall a'.\mathit{Occ}(a') \supset \neg \mathit{IsDomAct}(a')) \\ &\vee (n \neq 0 \wedge \neg \mathit{IsDomAct}(a) \wedge \mathit{PlanOrder} = n) \\ &\vee (\mathit{IsDomAct}(a) \wedge (\mathbf{V}\,\mathit{PlanOrder} = n - 1 \oplus n = 1))\end{aligned}$$

Formulas to constrain the time interval I in which the n-th plan action starts can be simply given as follows:

$$\mathfrak{abs}(n, I) := F_I(\mathit{Occ}(a(\vec{x})) \wedge \mathit{PlanOrder} = n \wedge \mathit{IsDomAct}(a))$$

Other than providing on the absolute bounds, where each action is started, the domain reasoning may also depend on the relative temporal relations between plan actions. Therefore, we also provide constraints to specify the time interval I in between the n-th and m-th plan action, with $m > n$:

$$\begin{aligned}\mathfrak{rel}(n, m, I) := \mathbf{F}\Big[&\mathit{Occ}(a(\vec{x}_1)) \wedge \mathit{PlanOrder} = n \wedge \mathit{IsDomAct}(a) \\ &\wedge \mathbf{F}_I\,\mathit{Occ}(a'(\vec{x}_2)) \wedge \mathit{PlanOrder} = m \wedge \mathit{IsDomAct}(a')\Big]\end{aligned}$$

4 Plan Synthesis as Reachability Problem

The difficulty of creating an executable plan in our setting stems from the way different interconnection constraints impact each other. To illustrate this, assume we are given a platform $\mathcal{A}$, a plan P that contains an action α_k with argument x and we have two interconnection constraints:

- $[\![\delta]\!]\mathbf{G}\left[\exists x.\mathrm{Occ}(\alpha_k(x)) \supset \mathbf{F}_{[0,30]}\mathrm{state}(\mathcal{A}) = s_i\right]$

  (Whenever α_k starts we want to reach state s_i within the next 30 seconds.)
- $[\![\delta]\!]\mathbf{G}\left[\exists x.\mathrm{Occ}(\alpha_k(x)) \supset \mathbf{F}_{[0,60]}\mathrm{state}(\mathcal{A}) = s_j\right]$

  (Whenever α_k starts we want to reach state s_j within the next 60 seconds.)

It is now unclear whether we should reach s_i before s_j or vice versa. So in general we cannot simply try to satisfy the constraints one by one but rather have to respect them simultaneously. To tackle this we developed different encoding strategies with varying success.

Our very first approach evolves purely around the notions of classical timed automata and can be described as a direct encoding of all possible decisions that a plan transformation may take in order to establish a trace suiting all constraints. While the idea is quite straightforward, the resulting automaton that forms the encoding is huge and requires a complex implementation, which makes debugging cumbersome. However, despite its simple nature the direct encoding significantly outperformed the other approaches we came up with, which is why we present it in detail in Section 4.1. One of the other attempts to find more elegant encodings that are easier to implement and debug is outlined in Section 4.2, although we found it to scale poorly compared to the direct encoding strategy.

4.1 Direct Encoding

A straightforward encoding for the plan transformation as proposed in the last section can be obtained, if the different aspects, namely the components, the plan and the constraints are combined into a single automaton. In the following we first explain the idea by showcasing an encoding on a simple example, before we give formal definitions on the different encoding steps.

4.1.1 Encoding Example

We consider the platform component $\mathcal{M}_{\texttt{vis}}$ from Figure 2.1, and a robot that performs grasping tasks. Let us begin with a constraint $\gamma_{\texttt{prep}}$, stating that the vision should be warming up 2 seconds prior to the execution of any `pick` or `put`:

$$\gamma_{\texttt{prep}} := \mathfrak{occ}(Occ(\texttt{pick}(o,p)) \vee Occ(\texttt{put}(o,p)), \mathbf{H}_{(0,2]}(state(\mathcal{M}_{\texttt{vis}}) = \texttt{warm-up} \vee state(\mathcal{M}_{\texttt{vis}}) = \texttt{running}))$$

We assume the domain reasoner to dispatch a plan

$$P_1 = \langle \texttt{pick}(BOOK, TABLE), \texttt{goto}(TABLE, SHELF) \rangle$$

together with temporal constraints stating that the plan may be started at any time and that it takes precisely 15 seconds to start `goto` after `pick`:

$$\gamma^1_{\texttt{abs}} := \mathfrak{abs}(1, [0, \infty))$$
$$\gamma^2_{\texttt{abs}} := \mathfrak{abs}(2, [15, \infty))$$
$$\gamma^1_{\texttt{rel}} := \mathfrak{rel}(1, 2, [15, 15])$$

Encoding a Plan The encoding we present evolves around different steps that lead to the construction of one automaton $\mathcal{A}_{\texttt{enc}}$ with a designated state s, such that every run reaching s corresponds to a timed trace satisfying the specified constraints of the form $\mathfrak{occ}, \mathfrak{uc}, \mathfrak{abs}$ and $\mathfrak{rel}$, while taking a plan $P = \langle a_1, \ldots, a_n \rangle$ as baseline. Let us denote the sets of constraints of the respective types by $C_{\mathfrak{occ}}, C_{\mathfrak{uc}}, C_{\mathfrak{abs}}$ and $C_{\mathfrak{rel}}$.

As a first step, we detail how to encode the temporal constraints concerned with high-level tasks into a timed automaton $\mathcal{A}_P$, such that any run on $\mathcal{A}_P$ that reaches s induces a timed trace of P satisfying all constraints in $C_{\mathfrak{abs}}$ and $C_{\mathfrak{rel}}$.

This is an easy task given that the only constraints concerning the plan actions are specified through intervals between different actions, we construct a timed automaton $\mathcal{A}_P$ as follows:

- $\mathcal{A}_P$ contains a state l_a for each action $a \in P$, as well as two designated states `start` and `fin`.
- Transitions between the locations l_a are given through the action sequence in P. Additionally, a transition from `start` to l_{a_1}, and from l_{a_n} to `fin` are added. So $\mathcal{A}_P$ is simply a line.
- A clock $x_{\texttt{abs}}$ is introduced to encode the constraints of $C_{\mathfrak{abs}}$.
- For each $\mathfrak{abs}(k, I_k) \in C_{\mathfrak{abs}}$ with (strict) upper bound w in I_k we add $x_{\texttt{abs}} \leq w$ ($x_{\texttt{abs}} < w$) as invariant to l_{k-1}. Similarly, the (strict) lower bound v in I_k is encoded as guard $x_{\texttt{abs}} \geq v$ ($x_{\texttt{abs}} > v$) on the incoming transition of l_k. We may omit the addition of the trivial bounds $x_{\texttt{abs}} \geq 0$ and $x_{\texttt{abs}} < \infty$.

Essentially, reaching a state l_{a_k} equates to starting a_k and the value of $x_{\texttt{abs}}$ when reaching l_{a_k} determines the time at which a_k starts. We explicitly encode the upper bounds as invariant on the predecessor states rather than also adding it as guard to the incoming transition of l_k for practical reasons: modern model checking tools like `UPPAAL` do not construct the full symbolic representation of the search space while solving model checking tasks such as reachability queries. Instead they utilize on-the-fly constructions of symbolic states [17] to only keep relevant symbolic paths within the memory. Having invariants on states of a preceeding state, instead of a guard on a transition that has to be taken later on any run reaching the state of interest may help to abort paths early.

- For each $\mathfrak{rel}(k, k', I) \in C_{\mathfrak{rel}}$ a new clock $x_{k,k'}$ is added that is reset on the incoming transition of l_{a_k}. Then a clock constraint encoding the lower bound of I is added to the guard on the incoming transition of $l_{a_{k'}}$ and the upper bound is added as invariant on all states l_i, $k \leq i < k'$.

  This may be optimized by using the same clock of different constraints $\mathfrak{rel}(k_1, k_1', I_1)$ and $\mathfrak{rel}(k_2, k_2', I_2)$, if if the clock usage does not overlap. In particular, there has to be no k_Δ, such that $k_1 < k_\Delta < k_1'$ and $k_2 < k_\Delta < k_2'$. For the remainder of this thesis we will use a designated clock $x_{\texttt{rel}}$ to encode all constraints of the form $\mathfrak{rel}(k, k+1, I)$ as they are guaranteed to never overlap.

Figure 4.1 shows the construction for our example plan P_1. All runs from `start` that reach the designated state `fin` in $\mathcal{A}_{P_1}$ induce a timed trace through the times at which the visited states are reached. An exemplary run with an initial clock assignment $\nu_0 : x_{\texttt{abs}} = x_{\texttt{rel}} = 0$ and where `fin` is reached with clock values $x_{\texttt{abs}} = 20$ and $x_{\texttt{rel}} = 15$ can be given as follows:

$$\langle \texttt{start}, \nu_0 \rangle \to \langle \texttt{start}, \nu_0 + 5 \rangle \to \langle \texttt{pick}_1, \underbrace{[\{x_{\texttt{rel}}\} \mapsto 0]\nu_0 + 5}_{:=\nu_1} \rangle$$
$$\to \langle \texttt{pick}_1, \nu_1 + 15 \rangle \to \langle \texttt{goto}_2, \nu_1 + 15 \rangle \to \langle \texttt{fin}, \nu_1 + 15 \rangle$$

Based on the value of $x_{\texttt{abs}}$ at the times when reaching `pick` and `goto` we can derive the timed trace $\langle (\texttt{pick}, 5), (\texttt{goto}, 15) \rangle$ as an abstract plan with grounded times that satisfy $C_{\mathfrak{abs}}$ and $C_{\mathfrak{rel}}$.

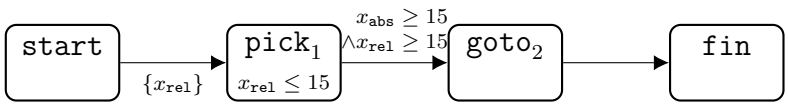


Figure 4.1: Plan P_1 converted to a timed automaton, action parameters are omitted.

Platform Control as Reachability Problem on Timed Automata Since the goal is to encode the whole plan transformation by means of runs reaching a

designated state s in some automaton $\mathcal{A}_{\texttt{enc}}$, we also face the problem to encode platform control tasks through reachability queries. We differentiate between two different types of basic control patterns:

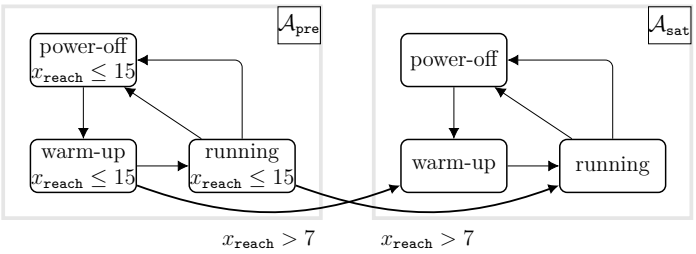
- Reach a state from a target set $L_{\texttt{target}}$ within given bounds I.
- Remain in a set of target states $L_{\texttt{target}}$ for given bounds I.

Beginning with the first task for a given platform automaton $\mathcal{A}_{\mathcal{M}}$, we note that the task is essentially composed out of three steps. Firstly, the platform is not constrained up to the lower bound of I, then at some point during I some target state may be reached, which is followed by unrestricted platform usage again. We depict the proposed construction in Figure 4.2a taking $\mathcal{M}_{\texttt{vis}}$ of Figure 2.1 as an example and proceed to explain them in more detail now:

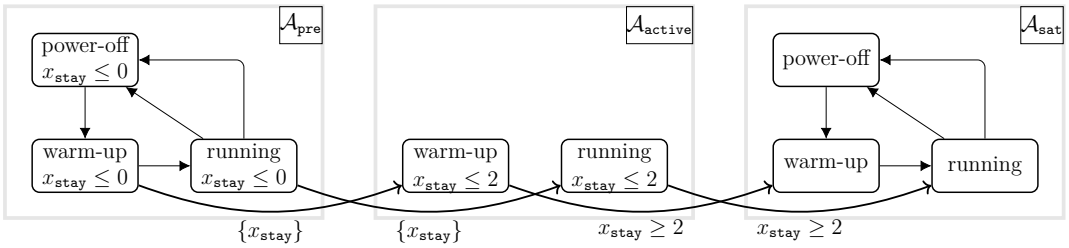
In order to encode those steps, we may create one copy $\mathcal{A}_{\texttt{pre}}$ of $\mathcal{A}_{\mathcal{M}}$ to represent the free usage of $\mathcal{M}$ prior to the event of reaching some state in $L_{\texttt{target}}$ after I and another copy $\mathcal{A}_{\texttt{sat}}$ to model the unrestricted platform usage after visiting $L_{\texttt{target}}$. The component (modeled through $\mathcal{A}_{\texttt{pre}}$) has to be in a one of the states from $L_{\texttt{target}}$ after enough time elapsed according to I, which should enable the move into the copy $\mathcal{A}_{\texttt{sat}}$. This can be modeled with the help of a fresh clock $x_{\texttt{reach}}$ by adding transitions from the locations of $L_{\texttt{target}}$ in $\mathcal{A}_{\texttt{pre}}$ to the respective copies in $\mathcal{A}_{\texttt{sat}}$ along with guard ensuring that $\mathcal{A}_{\texttt{sat}}$ may only be reached, if the lower bound is crossed on the valuation of $x_{\texttt{reach}}$. Similarly to the previous encoding of plan constraints we may encode the upper bound by adding invariants to $\mathcal{A}_{\texttt{pre}}$. We conclude that any run on the automaton as constructed above visits a state in $L_{\texttt{target}}$ on every run that starts in $\mathcal{A}_{\texttt{pre}}$ and reaches a state in $\mathcal{A}_{\texttt{sat}}$.

Addressing the other task, the need to remain $L_{\texttt{target}}$ for the time within I, we again provide an example construction that can be found in Figure 4.2b. We first note that in case I contains strict bounds, we can safely relax them to be non-strict, as the only way to remain in some states for the time within (v, w) is to already be there at v and only leave once the time is at least w. We may use the same trick as before to encode the given task, this time with three copies $\mathcal{A}_{\texttt{pre}}, \mathcal{A}_{\texttt{active}}$ and $\mathcal{A}_{\texttt{sat}}$. $\mathcal{A}_{\texttt{pre}}$ and $\mathcal{A}_{\texttt{sat}}$ again model the unrestricted control of $\mathcal{A}_{\mathcal{M}}$ before and after the stay in $L_{\texttt{target}}$, while $\mathcal{A}_{\texttt{active}}$ encodes the restriction to the target states in the time within I. Again, a fresh clock $x_{\texttt{stay}}$ is introduced to formulate guards and invariants in order to respect the relaxed time frame of I. The copies are connected via transitions from the states $L_{\texttt{target}}$ in $\mathcal{A}_{\texttt{pre}}$ to the respective copies in $\mathcal{A}_{\texttt{active}}$ and from there to the copies in $\mathcal{A}_{\texttt{pre}}$.

Combining Platform and Plan Automaton Now that we have the means to encode the high-level execution as reachability query via $\mathcal{A}_P$ and also know how to express control patterns of platform components in similar manner, it remains to combine both constructions, such that patterns according to the constraints from $C_{\texttt{occ}}$ and $C_{\texttt{uc}}$ can be encoded during the modeled execution of P within $\mathcal{A}_P$. To achieve this, the proposed procedure starts with the simplest case, where no



(a) Ensuring a visit of `warm-up` or `running` in $(7, 15]$ seconds.



(b) Remain in `warm-up` or `running` for 2 seconds.

Figure 4.2: Encoding Patterns to guide platform behavior. Dashed lines indicate successor transitions.

restrictions to $\mathcal{M}$ apply, as baseline encoding $\mathcal{A}_{\mathtt{bt}}$ that then gets extended step-wise for every constraint within $C_{\mathtt{occ}}$ and $C_{\mathtt{uc}}$.

Towards the construction of $\mathcal{A}_{\mathtt{bt}}$ we utilize the structure of the plan to divide the execution into the different time spans between the start of consecutive high-level actions. Therefore, we start with the automaton $\mathcal{A}_P$, which had the property that every run from `start` to `fin` corresponds to one of the timed trace of P satisfying the constraints $C_{\mathtt{abs}} \cup C_{\mathtt{rel}}$. The goal of $\mathcal{A}_{\mathtt{bt}}$ is to have an initial state, corresponding to the state `start` of $\mathcal{A}_P$ and the initial state of $\mathcal{A}_{\mathcal{M}}$, from which every run that reaches `fin` induces a valid trace of a transformed plan of P according to $C_{\mathtt{abs}} \cup C_{\mathtt{rel}}$. In particular, the runs may express every possible movement within $\mathcal{M}$ during P.

The construction essentially replaces each state in $\mathcal{A}_P$, except `fin`, by a copy of $\mathcal{A}_{\mathcal{M}}$, such that the copy replacing l_{a_k} corresponds to the possible platform operations between the start of the domain actions a_k and a_{k+1}, hence partitioning the continuous execution of $\mathcal{M}$ into finitely many separated pieces. The copy at `start` models the possible behavior of $\mathcal{M}$ before the first plan action is started. Required steps towards $\mathcal{A}_{\mathtt{bf}}$ given $\mathcal{A}_P = (L_p, \mathtt{start}, E_P, I_P)$ are shown below:

- For each state $l \in L_P \setminus \mathtt{fin}$ create a copy $\mathcal{A}_l$ of $\mathcal{A}_{\mathcal{M}}$. a_k and also add a designated state $\mathcal{A}_{\mathtt{fin}}$.
- For each state $l \in L_P \setminus \mathtt{fin}$ add the invariant $I_P(l)$ of l to each state in the corresponding $\mathcal{A}_l$.
- For each plan transition $e = l \xrightarrow{g,a,r} l' \in E_P$ add transitions from each state s in $\mathcal{A}_l$ to the corresponding copy s' in $\mathcal{A}_{l'}$ (or to `fin`, if l' corresponds to `fin` of $\mathcal{A}_P$) with the same annotations: $s \xrightarrow{g,a,r} s'$

Since the whole structure of $\mathcal{A}_P$ are carried over to $\mathcal{A}_{\mathtt{bt}}$, every run on $\mathcal{A}_{\mathtt{bt}}$ starting from the initial state of $\mathcal{A}_{\mathcal{M}}$ in the copy $\mathcal{A}_{\mathtt{start}}$ that reaches `fin` also induces a timed trace that satisfies the plan-specific constraints.

The construction of $\mathcal{A}_{\mathtt{bt}}$ is depicted schematically in Figure 4.3.

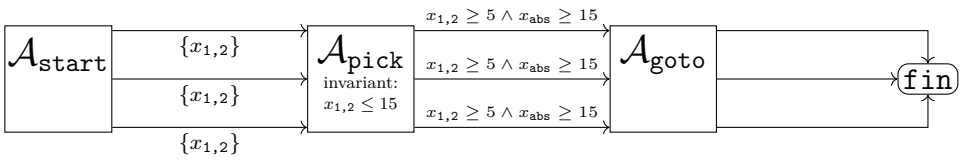


Figure 4.3: $\mathcal{A}_{P_1}$ and $\mathcal{A}_{\mathcal{M}_{\mathtt{vis}}}$ combined into $\mathcal{A}_{\mathtt{bt}}$.

Constraint Encoding Starting from $\mathcal{A}_{\mathtt{bt}}$, the constraints from $C_{\mathfrak{abs}} \cup C_{\mathfrak{rel}}$ can now be incorporated by utilizing patterns to control platform behavior, similar to the ones shown earlier. This is done by applying the following steps to each constraint $\gamma \in C_{\mathfrak{abs}} \cup C_{\mathfrak{rel}}$:

(1) Determining the domain actions that trigger the *activation* of γ (the need to manipulate the component).

(2) Calculating the *context* that γ influences (with respect to domain actions that may be started while γ enforces a certain behavior of the component).

(3) Encoding the possibilities to satisfy γ.

We start with the encoding of $\gamma_{\mathtt{prep}}$ and detail the necessary steps in the following:

The activations of $\gamma_{\mathtt{prep}}$ are obtained by looking at the occurrences of grasping actions in P. In our example plan P_1 this yields the first plan action `pick`, concluding (1).
$\gamma_{\mathtt{prep}}$ is concerned with the past two seconds relative to the actions where it gets active, indicated by the path operator $\mathbf{H}_{(0,2]}$ (Historically). We may utilize this information, by considering the partition of the execution span along the actions to determine during which of those intervals the platform operation has to take place. By imposing a strict lower bound in $\mathbf{H}_{(0,2]}$, the restrictions of $\gamma_{\mathtt{prep}}$ are applied only to all entries prior to the occurrence of `pick` in δ. With the construction of $\mathcal{A}_{\mathtt{bt}}$ in mind, we may conclude that the relevant context of $\gamma_{\mathtt{prep}}$ has to be covered by those automata copies within $\mathcal{A}_{\mathtt{bt}}$, that model the execution before `pick`, hence before the copy $\mathcal{A}_{\mathtt{pick}}$, and that cover the complete operation window of $\gamma_{\mathtt{prep}}$. For the given example plan P_1 this is simply the copy $\mathcal{A}_{\mathtt{start}}$, if the lower bound in $\mathbf{H}_{(0,2]}$ was non-strict instead, the copy $\mathcal{A}_{\mathtt{pick}}$ must have been taken into account as well as this would include the execution at the exact time of `pick` which is outside the scope of $\mathcal{A}_{\mathtt{start}}$.

By identifying the respective copies in $\mathcal{A}_{\mathtt{bt}}$ the task (2) is finished and we are left to encode the enforcement of $\gamma_{\mathtt{prep}}$ in the copy $\mathcal{A}_{\mathtt{start}}$. Since $\gamma_{\mathtt{prep}}$ demands that

$\mathcal{M}$ stays within $L_{\texttt{target}} = \{\texttt{warm-up}, \texttt{running}\}$ for two seconds prior to `pick` we may use a similar idea to the one shown in Figure 4.2b. The only difference is in fact, that while in Figure 4.2b the encoding assumes a future restriction to $L_{\texttt{target}}$ starting from $\mathcal{A}_{\texttt{pre}}$, we have to take care of a past encoding relative to $\mathcal{A}_{\texttt{pick}}$ at some time during $\mathcal{A}_{\texttt{start}}$. In fact, we could describe the encoding task as follows: At first, $\mathcal{M}$ may be operated freely according to $\mathcal{A}_{\texttt{start}}$, which we model through a copy $\mathcal{A}^{\texttt{prep}}_{\texttt{pre}}$ of $\mathcal{A}_{\texttt{start}}$. When the right time has come, the behavior of $\mathcal{M}$ according to $\mathcal{A}_{\texttt{start}}$ has to be restricted to the target states $L_{\texttt{target}}$, denoted by a copy $\mathcal{A}^{\texttt{prep}}_{\texttt{active}}$ of $\mathcal{A}_{\texttt{start}}$. Two seconds afterwards the time has to be equal to the activation start of `prep` and the restriction of $\gamma_{\texttt{start}}$ end, modeled by a third copy $\mathcal{A}^{\texttt{prep}}_{\texttt{sat}}$. This third copy essentially models the unrestrained control of $\mathcal{M}_{\texttt{vis}}$ once the time to start `put` has come, but is not carried out yet, allowing for traces such as:

$$\langle(\texttt{warm-up}, 0), (\texttt{no-op}, 2), (\texttt{shut-off}), 4), (\texttt{put,4})\rangle$$

We connect $\mathcal{A}^{\texttt{prep}}_{\texttt{pre}}, \mathcal{A}^{\texttt{prep}}_{\texttt{active}}$ and $\mathcal{A}^{\texttt{prep}}_{\texttt{sat}}$ similar as done in Figure 4.2b with transitions connecting the respective copy states of $L_{\texttt{target}}$. Utilizing of a fresh clock $x^1_{\texttt{prep}}$ we can encode the temporal relations between the three automata copies as follows:

- The clock is reset on the transitions to $A^{\texttt{prep}}_{\texttt{active}}$, essentially modeling a non-deterministic choice to move to $\mathcal{A}^{\texttt{prep}}_{\texttt{active}}$ in anticipation of the start of `pick` in two seconds.
- Invariants $x^1_{\texttt{prep}} \leq 2$ on each state in $\mathcal{A}^{\texttt{prep}}_{\texttt{active}}$ along with guards $\texttt{prep}^1 \geq 2$ on the transitions from $\mathcal{A}^{\texttt{prep}}_{\texttt{active}}$ to $\mathcal{A}^{\texttt{prep}}_{\texttt{sat}}$ ensure a stay in $\mathcal{A}^{\texttt{prep}}_{\texttt{active}}$ for exactly two seconds.
- Invariants $x^1_{\texttt{prep}} \leq 2$ on each state in $\mathcal{A}^{\texttt{prep}}_{\texttt{sat}}$ enforce that no time elapses in $\mathcal{A}^{\texttt{prep}}_{\texttt{sat}}$ as the incoming transitions carry the aforementioned guards $x^1_{\texttt{prep}} \geq 2$.

It remains to actually integrate the construction of $\mathcal{A}^{\texttt{prep}}_{\texttt{pre}}, \mathcal{A}^{\texttt{prep}}_{\texttt{active}}$ and $\mathcal{A}^{\texttt{prep}}_{\texttt{sat}}$ into $\mathcal{A}_{\texttt{bt}}$, which is a straightforward task: $\mathcal{A}_{\texttt{start}}$ is simply replaced by the three copies, incoming transitions into $\mathcal{A}_{\texttt{start}}$ instead connect to the states in $\mathcal{A}^{\texttt{prep}}_{\texttt{pre}}$ (since $A_{\texttt{start}}$ was the first copy within $\mathcal{A}_{\texttt{bt}}$ there are no incoming transitions), outgoing transitions from $\mathcal{A}_{\texttt{start}}$ to $\mathcal{A}_{\texttt{pick}}$ instead originate from $\mathcal{A}^{\texttt{prep}}_{\texttt{sat}}$. The resulting automaton $\mathcal{A}_{\texttt{enc}}$ can be found in Figure 4.4. An equivalent description of the encoding could state that two forks from the base timeline of $\mathcal{A}_{\texttt{bt}}$ at the sub-automaton $\mathcal{A}_{\texttt{pick}}$ were created in order to realize the eventual stay in $L_{\texttt{target}}$ followed by unrestrained operations, before starting `pick`.

Extending the Example In order to showcase the procedure of encoding even more constraints, we extend our example and consider the plan

$$P_2 = \langle \texttt{pick}(\mathit{BOOK},\ \mathit{TABLE}), \texttt{goto}(\mathit{TABLE},\ \mathit{SHELF}), \texttt{put}(\mathit{BOOK},\mathit{SHELF})\rangle$$

that picks up a book from a table and places it on a shelf, the reasoner imposes the following additional restrictions, stating that between `goto` and `put` $[30, 45)$

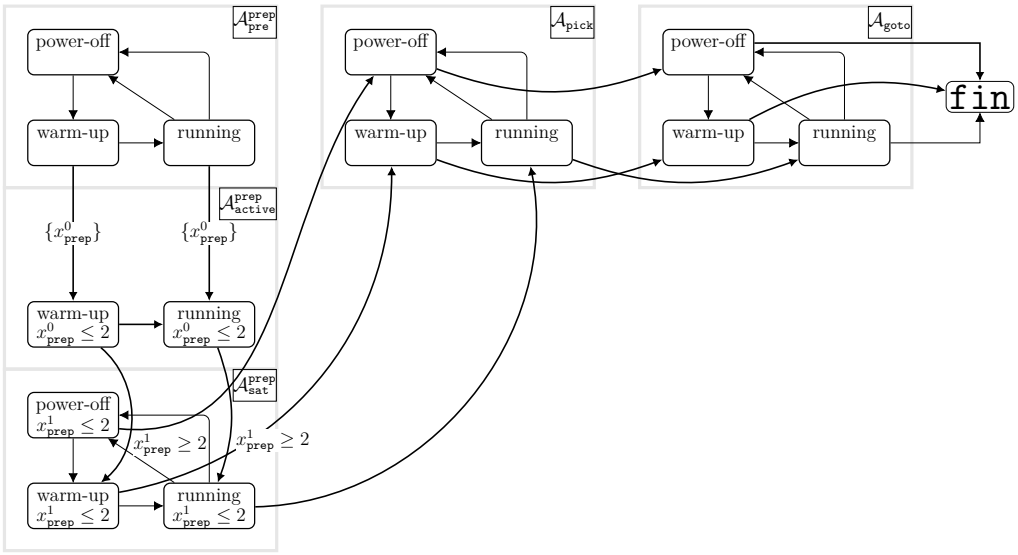


Figure 4.4: $\mathcal{A}_{\texttt{enc}}$ resulting from integration of $\gamma_{\texttt{pre}}$ into $\mathcal{A}_{\texttt{bt}}$ for the plan P_1. Only annotations towards the encoding of $\gamma_{\texttt{pre}}$ are shown.

seconds may elapse.

$$\gamma^3_{\texttt{abs}} := \mathfrak{abs}(2, [45, \infty))$$
$$\gamma^2_{\texttt{rel}} := \mathfrak{rel}(2, 3, [30, 45))$$

We assume the following constraints in addition to $\gamma_{\texttt{prep}}$:

$$\gamma_{\texttt{eco}} := \mathfrak{occ}(\mathit{Occ}(\texttt{goto}(s, d)), \mathbf{F}_{[0,3]}\mathit{state}(\mathcal{M}_{\texttt{vis}}) = \texttt{power\_off})$$
$$\gamma_{\texttt{data}} := \mathfrak{occ}(\mathit{Occ}(\texttt{pick}(o, p)) \vee \mathit{Occ}(\texttt{put}(o, p)), \mathit{state}(\mathcal{M}_{\texttt{vis}}) = \texttt{running}\mathbf{U}_{[10,10]}\top)$$

$\gamma_{\texttt{eco}}$ enforces the vision to be powered off shortly after starting a `goto` action and $\gamma_{\texttt{data}}$ requires the vision to be running for the first 10 seconds of each grasping action.

The plan automaton $\mathcal{A}_{P_2}$ is shown in figure Figure 4.5, from which we can perform the same steps as before to get $\mathcal{A}_{\texttt{bt}}$.

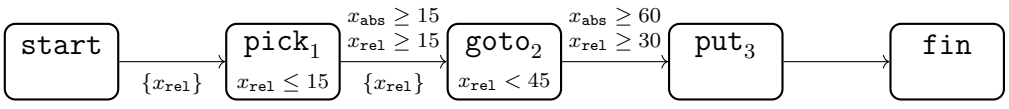


Figure 4.5: Plan P converted to a timed automaton, action parameters are omitted.

The encoding of $\gamma_{\texttt{prep}}$ is done analogously, however the constraint activates also due to `put`, requiring it to be encoded twice, once to replace $\mathcal{A}_{\texttt{start}}$ of $\mathcal{A}_{\texttt{bt}}$ and once to replace $\mathcal{A}_{\texttt{goto}}$.

Considering $\gamma_{\texttt{eco}}$ now we can follow the three steps presented before again: the activation is at the occurrence of `goto`, the context of $\gamma_{\texttt{prep}}$ is during the time

between `goto` and `pick`, because $\gamma^2_{\texttt{rel}}$ states that in the meantime at least 30 seconds elapse, while $\gamma_{\texttt{eco}}$ is only concerned with the following $[0, 3]$ seconds once `goto` starts (through $\mathbf{F}_{[0,3]}$). Therefore, he relevant time window of $\gamma_{\texttt{eco}}$ is covered by the three copies of $\mathcal{A}_{\mathcal{M}_{\texttt{vis}}}$ added through the previous encoding of $\gamma_{\texttt{prep}}$ to replace $\mathcal{A}_{\texttt{goto}}$.

The encoding is done in similar fashion: The operator $\mathbf{F}_{[0,3]}$ can be represented through the pattern from Figure 4.2a. In order to respect the bounds $[0, 3]$, a fresh clock $x_{\texttt{eco}}$ is introduced that has to be reset on transitions between the preceding `pick` and `goto`. Forks are created (representing $\mathcal{A}^{\texttt{eco}}_{\texttt{sat}}$) from the existing timelines (resembling $\mathcal{A}^{\texttt{eco}}_{\texttt{pre}}$), that get connected via copy transitions between the states `power-off`. The incoming and outgoing transition into/out of the original forks are instead applied to $\mathcal{A}^{\texttt{eco}}_{\texttt{pre}}$ and $A^{\texttt{eco}}_{\texttt{sat}}$. The result is an encoding of the time between `goto` and `put` via six copies of $\mathcal{A}_{\mathcal{M}_{\texttt{vis}}}$.

Lastly, the constraint activation $\gamma_{\texttt{data}}$ at `pick` can be encoded via two copies $\mathcal{A}^{\texttt{data}}_{\texttt{active}}$ and $\mathcal{A}^{\texttt{data}}_{\texttt{sat}}$ of $\mathcal{A}_{\texttt{pick}}$, one modeling the restriction to `running`, while the other allows for the possibility to freely operate $\mathcal{M}$ again 10 seconds after `pick` started. A fresh clock $x^1_{\texttt{data}}$ is introduced in order to track the time upon starting `pick` up to to the point, when the transition to $\mathcal{A}^{\texttt{data}}_{\texttt{sat}}$ may happen. The resulting automaton is shown in Figure 4.6.

Pitfalls and Optimizations The creation of automata copies to encode the plan transformation problem can lead to a huge resulting automaton, e.g., the encoding grows fast through constraints that

- are activated often,
- could be satisfied anywhere in a large context during the execution of high-level actions,
- require the introduction of a fresh clock for each activation and
- barely restrict the platform behavior, such that the added copies are roughly of the same size as the original platform automaton.

A constraint that is worst in the above sense can be easily given via:

$$\gamma_{\texttt{bad}} := \texttt{occ}(\bigvee_{a \in A} Occ(a(\vec{x}_a)), \mathbf{F}_{[0,v]} \bigvee_{l \in L_{\mathcal{M}} \setminus l_0} state(\mathcal{M}) = l),$$

where A denotes the finite set of all possible action standard names, v is an integer that is larger than the sum of all action duration lower bounds of entries in a high-level plan P and $\mathcal{M}$ is a platform model with states L.

The constraint $\gamma_{\texttt{bad}}$ essentially demands $\mathcal{M}$ to not be in some specified location l_0 somewhere during the scope of the plan after starting any domain action, hence

- it activates on every occasion during P,
- due to the choice of v the context of each activation is always the whole

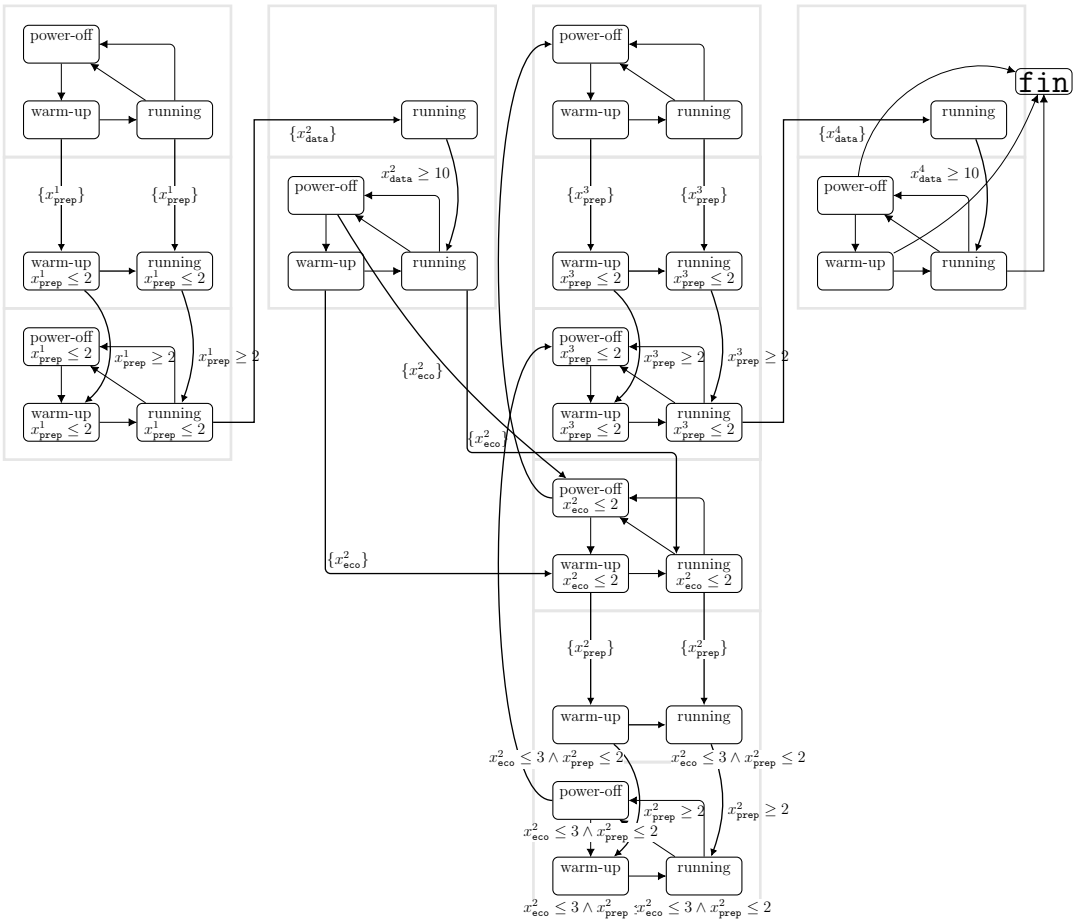


Figure 4.6: TA encoding plan, component and interconnection constraints. Only annotations concerning the encoding of $\gamma_{\texttt{prep}}$, $\gamma_{\texttt{eco}}$ and $\gamma_{\texttt{data}}$ are shown.

remaining plan,

- it requires a clock on each activation to track the distance to the respective activation time point and
- the copies that have to be created are of size $|L| - 1$.

An implementation without further optimizations would therefore create $|P|$ copies for the initial timeline, $|P|$ copies for the first activation, $2(|P| - 1)$ copies on the next activation (because the copies of the previous activations are copied again), followed by $4(|P| - 1)$ copies and so on, for a total of

$$|P| + \sum_{0 \leq i < |P|} 2^i (|P| - i) = 2^{n+1} - 2$$

copies with

$$\frac{2^{n+1} - 2}{2} |L| + \frac{2^{n+1} - 2}{2} (|L| - 1) = (2^n - 1)(2|L| - 1)$$

platform states just for the encoding of a single constraint. The considerations to craft worst-case examples may prove helpful when designing platform constraints

as it suggests that the plan transformation performance works best if the constraints are modeled with the following in mind:

- Constraints should activate only when really necessary.

  In a scenario, where $\gamma_{\texttt{bad}}$ would accurately model some platform behavior, a different angle on the situation may provide an equivalent constraint. $\gamma_{\texttt{bad}}$ essentially models that the l_0 should be avoided at some point during any plan execution, which may also be achieved by only activating it on the first occurring plan action, through introducing a designated dummy action that is appended to each plan.

- Constraints that significantly restrict the platform behavior through parts of the execution may help improve the resulting encoding.

- The encoding size heavily depends on the execution context of the constraints, keeping the context as short as possible may be beneficial for the plan transformation.

  Utilization of domain specific knowledge may help restraining platform information either through time bounds, by specifying intervals I on the available trace operators when using $\mathfrak{occ}$ constraints, or context based, by using until-chain constraints $\mathfrak{uc}$ when the platform operations are bounded by some high-level patterns.

Aside from appealing to the platform designers to be aware of the impact the modeled constraints have on the plan transformation encoding, some common optimizations may be considered during the construction of such encodings.

Pruning the Encoding A basic step that may greatly simplify the constructed model checking problem is to detect dead ends with respect to possible runs through the system that reach the designated final state `fin`. Automata copies that do not have any outgoing edges to other copies are obvious candidates that may be deleted recursively, because they cannot be visited on any run that reaches `fin`. Upon closer inspection of Figure 4.6 there is a dead end that can be spotted by inspecting the time windows of active constraints during the execution of a high-level action. During `goto` the case of satisfying $\gamma_{\texttt{prep}}$ before $\gamma_{\texttt{eco}}$ is encoded, which is infeasible, because `goto` takes at least 30 seconds and $\gamma_{\texttt{prep}}$ restricts the platform usage during the first 3 seconds while $\gamma_{\texttt{eco}}$ demands the behavior within the last 2 seconds, leaving a gap of at least 25 seconds. One may be able to detect those implicit dead ends by deeper analysis of the partitioned time intervals between the occurrence of domain actions, which we consider outside of the scope of this thesis.

Recycling Clocks Constraints that activate on multiple occasions during a run may not require the addition of a fresh clock on each activation. If the execution contexts of different activations do not overlap, the same clock may be reused

again, e.g., the clock $x^3_{\texttt{prep}}$ is not necessary in Figure 4.6, because the clock $x^1_{\texttt{prep}}$ can be reused. We note that not only the context where constraints activate has to be considered, but also the span from the domain actions triggering the activations. The necessity can be easily demonstrated through the following constraint:

$$\gamma_{\texttt{late}} := \mathfrak{occ}(Occ(\texttt{action}_1()), \mathbf{F}_{[35,40]}\, state(\mathcal{M}) = \texttt{loc}_1)$$

Given a plan $\langle \texttt{action}_1(), \texttt{action}_2(), \texttt{action}_1(), \texttt{action}_3(), \texttt{action}_3() \rangle$, where $\texttt{action}_1$ and $\texttt{action}_2$ take 10 seconds each and $\texttt{action}_3$ takes 20 seconds to execute, $\gamma_{\texttt{late}}$ is active during the first (second) execution of $\texttt{action}_3$ due to the trigger at the first (second) occurrence of $\texttt{action}_1$, as depicted in Figure 4.7. Even tho their active window is distinct, the span where the associated clocks track the time in order to satisfy the bound within $\gamma_{\texttt{late}}$ are overlapping. Therefore, two different clocks are required in order to encode the distinct activations of $\gamma_{\texttt{late}}$.

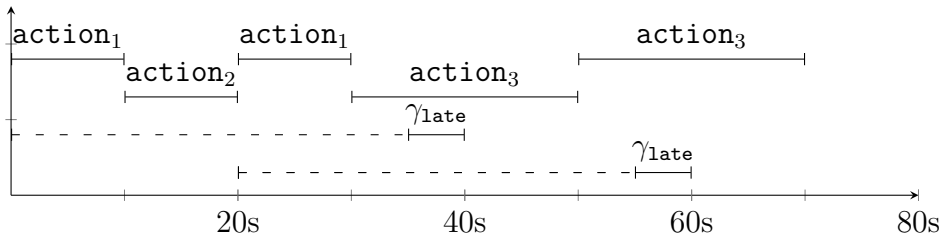


Figure 4.7: Activation duration versus clock usage windows.

4.1.2 Auxiliary Functions

Heading towards defining the formal steps necessary to encode the different constraints we begin with establishing common notations for the remainder of this section.

We denote the high-level plan to transform by P with $P = \langle p_0 = \texttt{start}, p_1, \ldots, p_n \rangle$ and denote the intervals describing the time frame between p_i to p_{i+1} by b_i, for $0 \leq i < n$. We also write $P_b = \langle (p_0, b_0), (p_1, b_1), \ldots, (p_n, b_n) \rangle$ to reference the action with the associated intervals. Those intervals can be obtained by looking at the constraints $C_{\mathfrak{abs}} \cup C_{\mathfrak{rel}}$. For the scope of this thesis we are going to assume that the high-level reasoner explicitly states all relations such that no further relations can be deduced, e.g., if $\mathfrak{abs}(1, [0, 4])$, $\mathfrak{abs}(2, [0, 10])$ and $\mathfrak{rel}(1, 2, [5, 5])$ holds, it is possible to deduce $\mathfrak{abs}(2, [5, 9])$, hence the reasoner should provide it. We also assume that trivial relations are also explicitly stated, such that for each $1 \leq k \leq n$ there exists a constraint $\mathfrak{abs}(k, I_k)$ ($I_k = [0, \infty)$ as default value) and for each $1 \leq k < n$, $\mathfrak{rel}(k, k+1, I_k)$ is also given (again with default value I_k). Therefore the bounds b_i can be given via:

- $b_0 = [0, w]$ ($b_0 = [0, w)$) for a (strict) upper bound of w in I_1 of $\mathfrak{abs}(1, I_1)$
- $b_k = I_k$ for $\mathfrak{rel}k, k+1, I_k$, for $1 \leq k < n$

- $b_n = [0, \infty)$

Further we refer to the platform model to control as $\mathcal{A}_\mathcal{M}$ with locations $L_\mathcal{M}$, the automaton forming the encoding is called $\mathcal{A}_{\texttt{enc}}$, initialized via $\mathcal{A}_{\texttt{enc}} = \mathcal{A}_{\texttt{bt}}$ (see Figure 4.3). We also utilize the following auxiliary functions:

- COPY$(\mathcal{A}, L_{\texttt{target}})$ creates a copy of an automaton $\mathcal{A}$ and deletes all states not included in $L_{\texttt{target}}$ from the copy. If no states should be deleted, then the second parameter is omitted.
- ADDINVARIANTS$(\mathcal{A}, \mathcal{A}', \phi)$ extends the invariants of all states in $\mathcal{A}'$ that is a sub-automaton of $\mathcal{A}$ to contain the clock constraint ϕ (concatenation with $\wedge$).
- ADDCOPYTRANS$(\mathcal{A}, \mathcal{A}', \widetilde{\mathcal{A}}, \phi, u, L_{\texttt{target}})$ adds copy transitions (as done in Figure 4.2) from $\mathcal{A}'$ to $\widetilde{\mathcal{A}}$ that are both sub-automata of $\mathcal{A}$. Additionally annotates the added transitions by a clock constraint ϕ and update set u. Optionally a state $L_{\texttt{target}}$ may be supplied, which then only adds transitions to $L_{\texttt{target}}$ within $\widetilde{\mathcal{A}}$.
- ADDSUCCTRANS$(\mathcal{A}, \mathcal{A}', \widetilde{\mathcal{A}}, \phi, u, L_{\texttt{target}})$ similar to ADDCOPYTRANS, but creates transitions that simultaneously move from $\mathcal{A}'$ to $\mathcal{A}'$, while taking a transition t within $\mathcal{A}_\mathcal{M}$. Hence the added transition also get the same annotations as t, extended by ϕ and u.

  Successor transitions are required if progression from one set of target states $L_{\texttt{target}}$ into another set $L'_{\texttt{target}}$, has to be enforced. This becomes necessary when encoding constraints of the form $\mathfrak{occ}(\beta, \alpha_1 \mathbf{U}_I \alpha_2)$ (progression from α_1 to α_2), $\mathfrak{occ}(\beta, \alpha_1 \mathbf{S}_I \alpha_2)$ (transitions from α_2 to α_1) and $\mathfrak{uc}(B, \beta_1, \beta_2)$ (along the sequence of α_i within B, if B is a sequence of length >1).
- TIMELINES$(\mathcal{A}, i, j)$ returns the sub-automaton within $\mathcal{A}_{\texttt{enc}}$ spanning from the execution after p_i start until after p_j, e.g., *timelines*$(0, 1)$ for the encoding in Figure 4.4 covers everything, except $fancyA_{\texttt{goto}}$ and `fin`.
- ADDTOINCTRANS$(\mathcal{A}, i, \phi, u)$ extends the clock constraints and updates of transitions within $\mathcal{A}$ that reach from the timeline forks during p_{i-1} to the ones during p_i.
- RESTRICTSTATES$(\mathcal{A}, \mathcal{A}', L_{\texttt{target}})$ deletes all states from the sub-automaton $\mathcal{A}'$ of $\mathcal{A}$ that are not contained in the target set $L_{\texttt{target}}$.
- SHIFTOUTGOINGTRANS$(\mathcal{A}, j, \mathcal{A}', \widetilde{\mathcal{A}})$ Changes the sources of transitions within $\mathcal{A}$ that originate in timeline forks during p_j in the sub-automaton $\mathcal{A}'$ to instead originate from the corresponding copy states in the sub-automaton $\widetilde{\mathcal{A}}$.

To further simplify the procedural descriptions, we establish some shorthand notations to describe clock constraints:

- SATUB(x, I) denotes $x < w$ for $I = \langle v, w)$ and $x \leq w$ for $I = \langle v, w]$. In the

special case of $I = \langle v, \infty)$ it equates to $\top$.

- UNSATLB(x, I) denotes $x \leq v$ for $I = (v, w\rangle$ and $x < v$ for $I = [v, w\rangle$.
- UNSATUB(x, I) denotes $x \geq w$ for $I = \langle v, w)$ and $x > w$ for $I = \langle v, w]$. In the special case of $I = \langle v, \infty)$ it equates to $\bot$.
- SATLB(x, I) denotes $x > v$ for $I = (v, w\rangle$ and $x \geq v$ for $I = [v, w\rangle$.
- In similar fashion we define BELOWOREQUB, ABOVEOREQUB, BELOWOREQLB and ABOVEOREQLB to always contain the non-strict comparison $\leq$ or $\geq$.

In order to calculate the context of a constraint activation, so the window of domain actions during which some behavior is enforced we utilize the operators from the algebra over bounds as described in Section 2. To that end we treat intervals I as tuple of bounds, e.g. $I = [v, w)$ consists of UB$(I) = \langle w, <\rangle$ and LB$(I) = \langle -v, \leq\rangle$ (negative value because the algebra was defined based on upper bounds). The context calculation for a constraint that imposes platform restrictions in the I_c from the activation at the start of some domain action p_k then boils down to the following two steps: Firstly, The first high-level action that may lay within the context of I_c is determined by accumulating the upper bounds of the durations of actions in P that get executed from p_k onward until the lower bound of I_c is crossed. This essentially skips through all plan actions that are guaranteed to be executed before I_c. As second step, the last action within P that may lay inside I_c is calculated by accumulating the lower bounds of action durations from p_k onward, until the upper bound of I_c is crossed. Any action afterwards is ensured to be outside of the context given through I_c. The procedure is depicted in Algorithm 1.

4.1.3 Encoding $\mathfrak{occ}(\beta, \mathbf{G}_I\alpha)$

A constraint of the form $\mathfrak{occ}(\beta, \mathbf{G}_I\alpha)$ requires $\mathcal{A}_\mathcal{M}$ to be within the target states $L_{\texttt{target}}$ defined by α for the interval I after the execution start of domain actions specified in β. This essentially equates to counting upwards from an activation at some plan action p_k, until the time has come (according to the interval I) to become active by remaining in the target states until the scope of I is passed. The scope within the plan may be expressed in terms of actions p_i and p_j in between which the time described by the interval I elapses. A Schematic drawing given in Figure 4.8 visualizes the proposed transformation. The necessary steps upon encountering an activation of $\mathfrak{occ}(\beta, \mathbf{G}_I\alpha)$ at a domain action p_k are given in the following and can be summed up to the algorithm in Algorithm 2:

1. The first step is to calculate the context $\langle i, j\rangle$ from p_k with respect to I via Algorithm 1. The copies of $\mathcal{A}_\mathcal{M}$ that lay within the range between p_i and p_j are those, from which it may become necessary to move to the target states.
2. A clock $x_{\texttt{clock}}$ is introduced to the current encoding $\mathcal{A}_{\texttt{enc}}$ that gets reset on

Algorithm 1 CALCULATECONTEXT(k, I, P) Calculate Context of Constraint Activation within a Plan

Require: $P_b = \langle (p_0, b_0), (p_1, b_1), \ldots, (p_n, b_n) \rangle$, $\texttt{start}$, b_c
Ensure: $\langle i, j \rangle$ ▷ Context within P from $p_{\texttt{start}} + b_c$

```
i, j ← start
lb_acc ← ⟨0, ≤⟩
ub_acc ← ⟨0, ≤⟩
for all (p_k, b_k) in ⟨(p_start, b_start), ..., (p_n, b_n)⟩ do
    lb_acc ← lb_acc + LB(b_k)
    ub_acc ← ub_acc + UB(b_k)
    if ub_acc < |LB(b_c)| then
        i ← i + 1
    end if
    j ← j + 1
    if |lb_acc| > UB(b_c) then
        return ⟨i, j⟩
    end if
end for
return ⟨i, j⟩
```

the incoming transitions to the forks at the execution of p_k. This initiates the counter to reach the range within I from p_k onward.

3. Then we create copy $\mathcal{A}_{\texttt{active}}$ of the context $\langle i, j \rangle$ that restricts usage to the target states. They resemble the need to actually remain in the target states $L_{\texttt{target}}$ somewhere between p_i and p_j due to $x_{\texttt{clock}}$ crossing the lower bound of I.

4. Yet another copy $\mathcal{A}_{\texttt{sat}}$ of the context $\langle i, j \rangle$ then models the execution after the activation of $\mathfrak{occ}(\beta, \mathbf{G}_I \alpha)$ is satisfied.

   Here we can reapply the idea of an execution context again: The first action p_i of the context represents the earliest action during which $\mathfrak{occ}(\beta, \mathbf{G}_I \alpha)$ possibly enforces the stay in $L_{\texttt{target}}$ up to the point when $c_{\texttt{clock}}$ crosses w. This also implies that from p_i on a minimum time of $w - v$ may elapse before $\mathcal{A}_{\texttt{sat}}$ can be reached. So instead of copying the whole context of $\langle i, j \rangle$, we can replace i by the lower end of the context starting at i with interval $[w - v, w - v]$.

5. We then have to establish connections from the original context over $\mathcal{A}_{\texttt{target}}$ to $\mathcal{A}_{\texttt{sat}}$, which we may do by utilizing copy transitions.

6. Utilizing clock constraints on transition guards and invariants we can constraining the system such that any valuation of the clock $x_{\texttt{clock}}$ after p_k that lies within I requires the current state to be within the target states, hence in $\mathcal{A}_{\texttt{active}}$.

   Because the trace operator $\mathbf{G}$ requires the system to be in $L_{\texttt{target}}$ for the

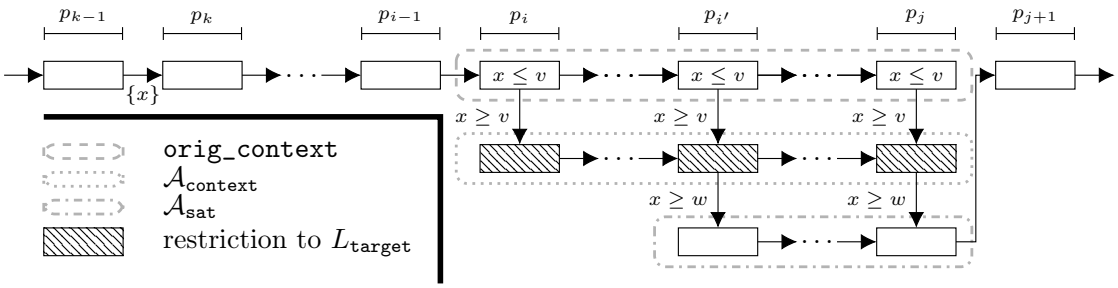


Figure 4.8: Encoding of $\mathfrak{occ}(\beta, \mathbf{G}_I\alpha)$, with a clock x to constrain the platform usage to an interval $I = (v, w)$ relative to the start of p_k. I describes a duration covered between the high-level actions p_i and p_k (`orig_context`) during which at some point x crosses v at which point the target states have to be reached ($\mathcal{A}_{\texttt{context}}$). the platform has to remain there until x crosses the upper bound, at which point normal execution may continue ($\mathcal{A}_{\texttt{sat}}$).

whole duration of I we have to relax any strict bound to a non-strict one within I when formulating the clock constraints. This is due to the fact that time can only elapse in states, hence the only way to satisfy the semantics of $\mathbf{G}$ formulated on strict bounds is, to already (still) be in $L_{\texttt{target}}$ at the sharp endpoints of I.

7. Lastly we ensure a stay in $\mathcal{A}_{\texttt{active}}$ on any run in $\mathcal{A}_{\mathcal{M}_{\texttt{enc}}}$ that reaches `fin` by moving outgoing transitions of the timeline forks during p_j to the respective copies in $\mathcal{A}_{\texttt{sat}}$.

Algorithm 2 ENCODE($\mathfrak{occ}(\beta, \mathbf{G}_I\alpha)$) at occurrence p_k

Require: $k, x_{\texttt{clock}}, \mathcal{A}_{\texttt{enc}}, L_{\texttt{target}}, I = \langle v, w \rangle, P_b$
Ensure: $\mathcal{A}'_{\texttt{enc}}$ ▷ Encoding of $\mathfrak{occ}(\beta, \mathbf{G}_I\alpha)$ at p_k into $\mathcal{A}_{\texttt{enc}}$

1: $\langle i, j \rangle \leftarrow$ CALCULATECONTEXT(k, I, P_b)
2: `orig_context` $\leftarrow$ TIMELINES$(\mathcal{A}_{\texttt{enc}}, i, j)$
3: $\mathcal{A}_{\texttt{sat}} \leftarrow$ COPY(`orig_context`)
4: $\langle i', j' \rangle \leftarrow$ CALCULATECONTEXT$(i, [w - v, w - v], P_b)$
5: $\mathcal{A}_{\texttt{active}} \leftarrow$ COPY(TIMELINES$(\mathcal{A}_{\texttt{enc}}, i', j), L_{\texttt{target}}$)
6: $\mathcal{A}'_{\texttt{enc}} \leftarrow \mathcal{A}_{\texttt{enc}} \cup \mathcal{A}_{\texttt{active}} \cup \mathcal{A}_{\texttt{sat}}$
7: ADDTOINCTRANS$(\mathcal{A}'_{\texttt{enc}}, k, \top, \{x_{\texttt{clock}}\})$
8: ADDINVARIANTS$(\mathcal{A}'_{\texttt{enc}},$ `orig_context`$,$ BELOWOREQLB$(x_{\texttt{clock}}, I))$
9: ADDCOPYTRANS$(\mathcal{A}'_{\texttt{enc}},$ `orig_context`$, \mathcal{A}_{\texttt{active}},$ ABOVEOREQLB$(x_{\texttt{clock}}, I), \emptyset)$
10: ADDCOPYTRANS$(\mathcal{A}'_{\texttt{enc}}, \mathcal{A}_{\texttt{active}}, \mathcal{A}_{\texttt{sat}},$ ABOVEOREQUB$(x_{\texttt{clock}}, I), \emptyset)$
11: SHIFTOUTGOINGTRANS$(\mathcal{A}'_{\texttt{enc}}, j,$ `orig_context`$, \mathcal{A}_{\texttt{sat}})$
12: **return** $\mathcal{A}'_{\texttt{enc}}$

4.1.4 Encoding $\mathfrak{occ}(\beta, \alpha_1 \mathbf{U}_I \alpha_2)$

The previous constraint type essentially describe two fixed time intervals through the attached interval $I = \langle v, w \rangle$, the duration up to v, where no restriction is imposed yet and then the duration from v to b, where the platform is restricted to the target states. Constraints of the form $\mathfrak{occ}(\beta, \alpha_1 \mathbf{U}_I \alpha_2)$ are only concerned with the enforcement of α_1 up to some point during I, leading to a comparably small encoding. The idea is exactly the same: Using a clock to count up to the time within I we can enforce the transition from the target states of α_1 to α_2 at the appropriate times. Actions p_i and p_j are determined to scope the position of I relative to the domain actions. The procedure is illustrated in Figure 4.9 and we

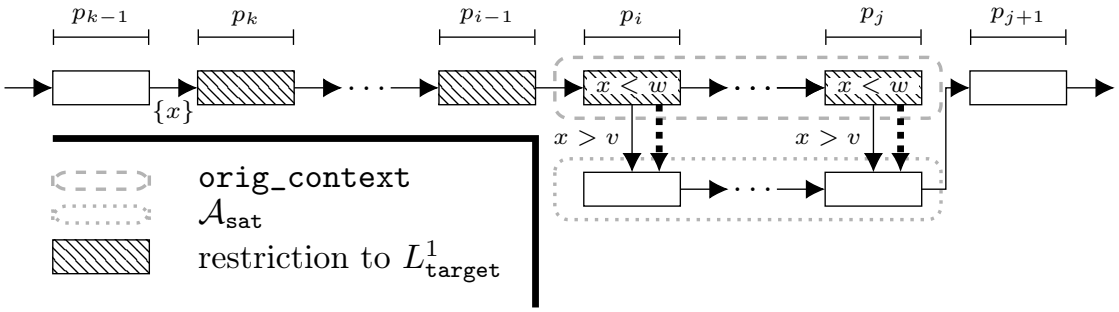


Figure 4.9: Encoding of $\mathfrak{occ}(\beta, \alpha_1 \mathbf{U}_I \alpha_2)$ for an interval $I = (v, w)$ and using a clock x from the occurrence of p_k onward. Platform control is restricted to α_1, until x reaches the range if I, which happens somewhere between the high-level actions p_i and p_j (`orig_context`). From there on, the target states of α_1 may be left by transitioning into a state from α_2 ($\mathcal{A}_{\mathtt{sat}}$). Thick dashed edges indicate successor transitions.

again give the steps towards an algorithm that can be found in Algorithm 3. The necessary steps are:

1. Calculating the context from k with respect to I.
2. Introducing a clock $x_{\mathtt{clock}}$ to $\mathcal{A}_{\mathtt{enc}}$ that gets reset on the incoming transitions to the forks at the execution of p_k
3. Creating a copy $\mathcal{A}_{\mathtt{sat}}$ of the context $\langle i, j \rangle$.
4. Restricting the original context from k to j to the target states described through α_1.
5. Adding copy and successor transitions to connect the forks from i to j to the respective copies in $\mathcal{A}_{\mathtt{sat}}$.
6. Constraining those transitions to require a valuation of $x_{\mathtt{clock}}$ within I.
7. Enforcing a visit in $\mathcal{A}_{\mathtt{sat}}$ on any run in $\mathcal{A}_{\mathcal{M}_{\mathtt{enc}}}$ that reaches `fin` by moving outgoing transitions of the timeline forks during p_j to the respective copies in $\mathcal{A}_{\mathtt{sat}}$.

The encoding of constraints $\mathfrak{occ}(\beta, \mathbf{F}_I\alpha)$ follows the exact scheme as $\mathbf{F}_I\phi = \top\mathbf{U}_I\phi$ and therefore is also covered.

Algorithm 3 ENCODE($\mathfrak{occ}(\beta, \alpha_1\mathbf{U}_I\alpha_2)$) at occurrence p_k

Require: $k, x_{\mathtt{clock}}, \mathcal{A}_{\mathtt{enc}}, L_{\mathtt{target}}$, $I = \langle v, w\rangle$, P_b
Ensure: $\mathcal{A}'_{\mathtt{enc}}$ ▷ Encoding of $\mathfrak{occ}(\beta, \alpha_1\mathbf{U}_I\alpha_2)$ at p_k into $\mathcal{A}_{\mathtt{enc}}$

1: $\langle i, j\rangle \leftarrow$ CALCULATECONTEXT(k, I, P_b)
2: $\mathtt{orig\_context} \leftarrow$ TIMELINES$(\mathcal{A}_{\mathtt{enc}}, i, j)$
3: $\mathcal{A}_{\mathtt{sat}} \leftarrow$ COPY$(\mathtt{orig_context})$
4: $\mathcal{A}'_{\mathtt{enc}} \leftarrow \mathcal{A}_{\mathtt{enc}} \cup \mathcal{A}_{\mathtt{sat}}$
5: ADDTOINCTRANS$(\mathcal{A}'_{\mathtt{enc}}, k, \top, \{x_{\mathtt{clock}}\})$
6: RESTRICTSTATES$(\mathcal{A}'_{\mathtt{enc}}, \mathtt{orig_context}, L^1_{\mathtt{target}})$
7: ADDINVARIANTS$(\mathcal{A}'_{\mathtt{enc}}, \mathtt{orig_context}, \mathrm{SATUB}(x_{\mathtt{clock}}, I))$
8: ADDCOPYTRANS$(\mathcal{A}'_{\mathtt{enc}}, \mathtt{orig_context}, \mathcal{A}_{\mathtt{sat}}, \mathrm{SATLB}(x_{\mathtt{clock}}, I), \emptyset, L^2_{\mathtt{target}})$
9: ADDSUCCTRANS$(\mathcal{A}'_{\mathtt{enc}}, \mathtt{orig_context}, \mathcal{A}_{\mathtt{sat}}, \mathrm{SATLB}(x_{\mathtt{clock}}, I), \emptyset, L^2_{\mathtt{target}})$
10: SHIFTOUTGOINGTRANS$(\mathcal{A}'_{\mathtt{enc}}, j, \mathtt{orig_context}, \mathcal{A}_{\mathtt{sat}})$
11: **return** $\mathcal{A}'_{\mathtt{enc}}$

4.1.5 Encoding $\mathfrak{occ}(\beta, \mathbf{H}_I\alpha)$

Encoding the past as required by $\mathfrak{occ}(\beta, \mathbf{H}_I\alpha)$ involves similar steps as they are used to construct the encoding of the counterpart constraints $\gamma_{\mathbf{G}}$, the two main differences being that the context calculation has to be adapted and that the clock encoding qualitative temporal information of I is used in different way:

Constraints concerned with the future reset a clock to zero and then treat the current clock value as a counter towards the interval of interest, while constraints that specify past behavior rather guess the correct time to reset a clock, indicating that the upper bound of the target interval relative to the activation at p_k is reached, and then count up towards the present.

Towards the necessary adaptions of CALCULATECONTEXT we observe that the basic procedure remains to be the calculation of those domain actions, during which a specified interval I (relative to a current action p_k) is located, by accumulating over the given action durations. The main difference is that the plan has to traversed backwards from p_k. The duration of p_k itself has to excluded, because only the past prior to the execution start of p_k is of interest.

Now it is a straightforward task to adapt Algorithm 2 in order to handle $\mathfrak{occ}(\beta, \mathbf{H}_I\alpha)$, the same steps apply, but the clock constraints and the transitions that update the clock change. Comparing Figure 4.8 with the schematic depiction to encode $\mathfrak{occ}(\beta, \mathbf{H}_I\alpha)$ that can be seen in Figure 4.10, we note that they essentially are a mirror image of each other. Hence the algorithm depicted in Algorithm 5 follows similar steps.

Algorithm 4 CALCULATECONTEXT′($\texttt{Start}, I, P_b$) Calculate Past Context of Constraint Activation within a Plan

Require: $P_b = \langle (p_0, b_0), (p_1, b_1), \ldots, (p_n, b_n) \rangle$, $\texttt{start}$, b_c
Ensure: $\langle i, j \rangle$ ▷ Context within P_b from $p_{\texttt{start}}$ - b_c
1: $P' \leftarrow \langle (p_{\texttt{start}-1}, b_{\texttt{start}-1}), (p_{\texttt{start}-2}, b_{\texttt{start}-2}), \ldots, (p_0, b_0) \rangle$
2: $\langle i, j \rangle \leftarrow$ CALCULATECONTEXT$(0, b_c, P')$
3: $\langle i, j \rangle \leftarrow \langle \texttt{start} - j - 1, \texttt{start} - i - 1 \rangle$
4: **return** $\langle i, j \rangle$

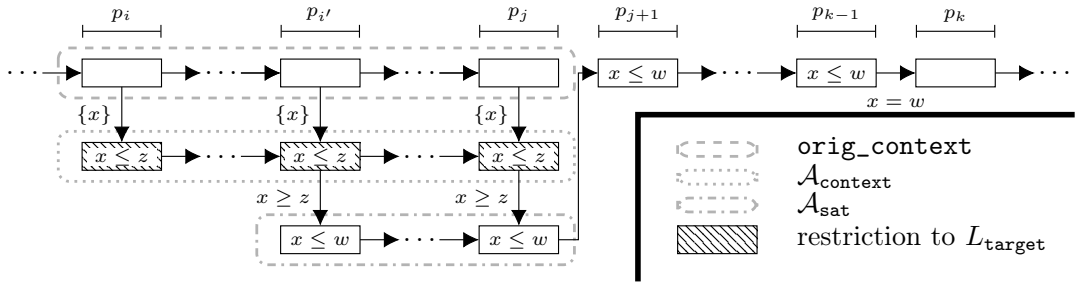


Figure 4.10: Encoding of $\mathfrak{occ}(\beta, \mathbf{H}_I \alpha)$ for an interval $I = (v, w)$ and a clock x at the activation of p_k. I describes a time prior to p_k that is bound between the start of actions p_i and p_k ($\texttt{orig\_context}$). From there on x gets reset and the target states of α are visited for the duration spanned by I ($\mathcal{A}_{\texttt{context}}$). Afterwards the component may be freely operated ($\mathcal{A}_{\texttt{sat}}$) until p_k is reached with x having the upper bound w of I as value.

4.1.6 Encoding $\mathfrak{occ}(\beta, \alpha_1 \mathbf{S}_I \alpha_2)$

In very similar manner we can adapt Algorithm 3 to obtain the encoding of constraints $\mathfrak{occ}(\beta, \alpha_1 \mathbf{S}_I \alpha_2)$ as the operator $\mathbf{S}$ is essentially the counterpart for reasoning about the past to the operator $\mathbf{U}$. Algorithm 6 depicts the necessary step towards an encoding of $\mathfrak{occ}(\beta, \alpha_1 \mathbf{S}_I \alpha_2)$ and Figure 4.11 provides a graphical illustration of the procedure.

4.1.7 Encoding $\mathfrak{uc}(B, \beta_1, \beta_2)$

Until-chain constraints $\mathfrak{uc}(B, \beta_1, \beta_2)$ can be encoded by applying steps from the construction of constraints with operator $\mathbf{U}$ (Section 4.1.4) for every entry of B. The context within the plan moves according the lower bounds on the respective intervals within B, hence the resulting encoding has a stairs-like shape of the execution windows p_i^r, p_j^r from the subsequent entries (α_r, I_r) within B, as illustrated in Figure 4.12. We also note that it suffices to use a single clock to encode every activation of $\mathfrak{uc}(B, \beta_1, \beta_2)$, because due to its definition to only span the shortest distance between the patterns β_1 and β_2, it is not possible to have two activations

Algorithm 5 ENCODE($\mathfrak{occ}(\beta, \mathbf{H}_I\alpha)$) at occurrence p_k

Require: $k, x_{\mathtt{clock}}, \mathcal{A}_{\mathtt{enc}}, L_{\mathtt{target}}, I = \langle v, w\rangle, P_b$
Ensure: $\mathcal{A}'_{\mathtt{enc}}$ ▷ Encoding of $\mathfrak{occ}(\beta, \mathbf{H}_I\alpha)$ at p_k into $\mathcal{A}_{\mathtt{enc}}$

1: $\langle i, j\rangle \leftarrow$ CALCULATECONTEXT$'(k, I, P_b)$
2: $\mathtt{orig\_context} \leftarrow$ TIMELINES$(\mathcal{A}_{\mathtt{enc}}, i, j)$
3: $\mathcal{A}_{\mathtt{sat}} \leftarrow$ COPY$(\mathtt{orig_context})$
4: $\langle i', j'\rangle \leftarrow$ CALCULATECONTEXT$'(i, [w - v, w - v], P_b)$
5: $\mathcal{A}_{\mathtt{active}} \leftarrow$ COPY$($TIMELINES$(\mathcal{A}_{\mathtt{enc}}, i', j), L_{\mathtt{target}})$
6: $\mathcal{A}'_{\mathtt{enc}} \leftarrow \mathcal{A}_{\mathtt{enc}} \cup \mathcal{A}_{\mathtt{active}} \cup \mathcal{A}_{\mathtt{sat}}$
7: ADDTOINCTRANS$(\mathcal{A}'_{\mathtt{enc}}, k, x_{\mathtt{clock}} = w, \emptyset)$
8: ADDINVARIANTS$(\mathcal{A}'_{\mathtt{enc}}, \mathcal{A}_{\mathtt{sat}},$ BELOWOREQUB$(x_{\mathtt{clock}}, I))$
9: ADDINVARIANTS$(\mathcal{A}'_{\mathtt{enc}},$ TIMELINES$(\mathcal{A}'_{\mathtt{enc}}, j + 1, k - 1),$ BELOWOREQUB$(x_{\mathtt{clock}}, I))$
10: ADDCOPYTRANS$(\mathcal{A}'_{\mathtt{enc}}, \mathtt{orig_context}, \mathcal{A}_{\mathtt{active}}, \top, \{x_{\mathtt{clock}}\})$
11: ADDCOPYTRANS$(\mathcal{A}'_{\mathtt{enc}}, \mathcal{A}_{\mathtt{active}}, \mathcal{A}_{\mathtt{sat}},$ ABOVEOREQUB$(x_{\mathtt{clock}}, I - v), \emptyset)$
12: SHIFTOUTGOINGTRANS$(\mathcal{A}'_{\mathtt{enc}}, j, \mathtt{orig_context}, \mathcal{A}_{\mathtt{sat}})$
13: **return** $\mathcal{A}'_{\mathtt{enc}}$

Algorithm 6 ENCODE($\mathfrak{occ}(\beta, \alpha_1\mathbf{S}_I\alpha_2)$) at occurrence p_k

Require: $k, x_{\mathtt{clock}}, \mathcal{A}_{\mathtt{enc}}, L_{\mathtt{target}}, I = \langle v, w\rangle, P_b$
Ensure: $\mathcal{A}'_{\mathtt{enc}}$ ▷ Encoding of $\mathfrak{occ}(\beta, \alpha_1\mathbf{S}_I\alpha_2)$ at p_k into $\mathcal{A}_{\mathtt{enc}}$

1: $\langle i, j\rangle \leftarrow$ CALCULATECONTEXT$'(k, I, P_b)$
2: $\mathtt{orig\_context} \leftarrow$ TIMELINES$(\mathcal{A}_{\mathtt{enc}}, i, j)$
3: $\mathcal{A}_{\mathtt{sat}} \leftarrow$ COPY$(\mathtt{orig_context})$
4: $\mathcal{A}'_{\mathtt{enc}} \leftarrow \mathcal{A}_{\mathtt{enc}} \cup \mathcal{A}_{\mathtt{sat}}$
5: ADDTOINCTRANS$(\mathcal{A}'_{\mathtt{enc}}, k,$ SATLB$(x_{\mathtt{clock}}, I), \emptyset)$
6: RESTRICTSTATES$(\mathcal{A}'_{\mathtt{enc}}, \mathtt{orig_context}, L^1_{\mathtt{target}})$
7: ADDINVARIANTS$(\mathcal{A}'_{\mathtt{enc}}, \mathcal{A}_{\mathtt{sat}},$ SATUB$(x_{\mathtt{clock}}, I))$
8: ADDINVARIANTS$(\mathcal{A}'_{\mathtt{enc}},$ TIMELINES$(\mathcal{A}'_{\mathtt{enc}}, j + 1, k - 1),$ SATUB$(x_{\mathtt{clock}}, I))$
9: ADDCOPYTRANS$(\mathcal{A}'_{\mathtt{enc}}, \mathtt{orig_context}, \mathcal{A}_{\mathtt{sat}}, \top, \{x_{\mathtt{clock}}\}, L^2_{\mathtt{target}})$
10: ADDSUCCTRANS$(\mathcal{A}'_{\mathtt{enc}}, \mathtt{orig_context}, \mathcal{A}_{\mathtt{sat}}, \top, \{x_{\mathtt{clock}}\}, L^2_{\mathtt{target}})$
11: SHIFTOUTGOINGTRANS$(\mathcal{A}'_{\mathtt{enc}}, j, \mathtt{orig_context}, \mathcal{A}_{\mathtt{sat}})$
12: **return** $\mathcal{A}'_{\mathtt{enc}}$

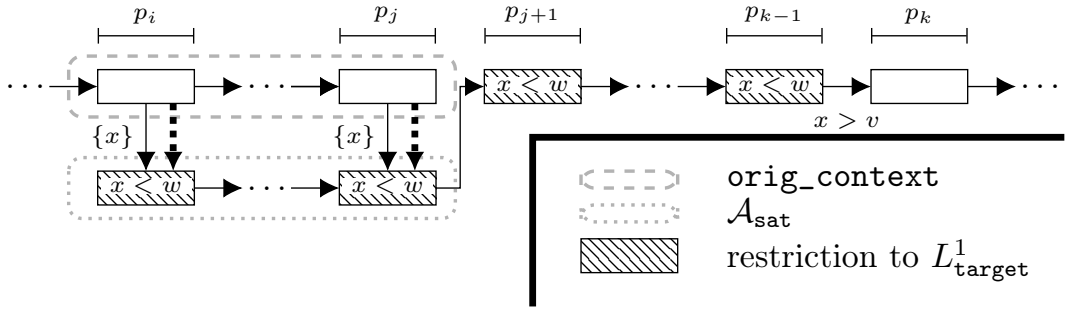


Figure 4.11: Encoding of $\mathfrak{occ}(\beta, \alpha_1 \mathbf{S}_I \alpha_2)$ for an interval $I = (v, w)$ and a fresh clock x at an activation p_k. Prior to p_k, bound by domain actions p_i and p_k (`orig_context`) the interval I relative to the past of p_k is located. Hence α_2 has to be visited (transitions from the states described by α_2 within `orig_context` to $\mathcal{A}_{\mathtt{sat}}$) which should be followed by a stay in the states specified by α_1 ($\mathcal{A}_{\mathtt{sat}}$) until p_k, at which point x is valuated within I. Thick dashed edges indicate successor transitions.

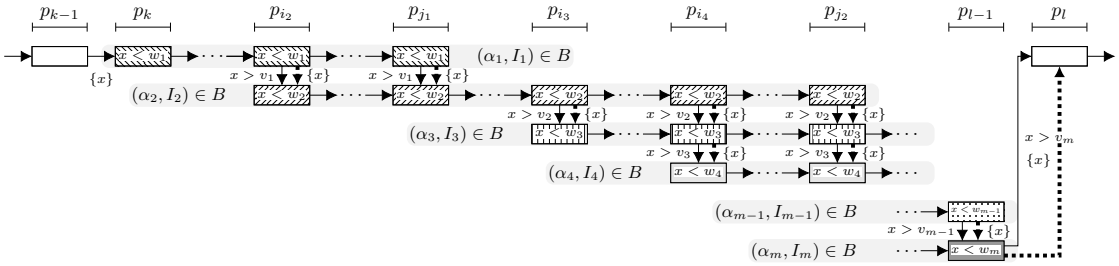


Figure 4.12: Encoding of $\mathfrak{uc}(B, \beta_1, \beta_2)$ for intervals $I_r = (v_r, w_r)$ using a clock x starting from p_k and ranging to p_l. B denotes a sequence of target state sets $L^r_{\mathtt{target}}$ that have to be visited for a duration I_r that lays between actions p^r_i and p^r_j relative to the plan. Hence a stairway pattern of transitions into subsequent target states is visited until p_l finally starts. Thick dashed edges indicate successor transitions.

of $\mathfrak{uc}(B, \beta_1, \beta_2)$ that overlap. Let $x_{\mathtt{clock}}$ denote the clock that we use for the remainder of this section. For $B = \langle (\alpha_1, I_1), (\alpha_2, I_2), \ldots, (\alpha_m, I_m) \rangle$ and a pattern from p_k to p_l matching β_1 and β_2 the procedure can be described as follows:

1. Reset $x_{\mathtt{clock}}$ on the incoming transitions to the forks at the execution of p_k.
2. Create $m-1$ copies $\mathcal{A}^r_{\mathfrak{uc}}, 1 < r \leq m$ of the forks spanning from p_k to p_{l-1} that we denote by $\mathcal{A}^1_{\mathfrak{uc}}$. The context $\langle i_r, j_r \rangle$ that each copy has to cover within the span to p_k to p_{l-1} can be be calculated my accumulating over the intervals that were covered so far, e.g., getting the context of $\sum_{t \leq r} I_t$ relative to p_k.
3. Restrict $\mathcal{A}^i_{\mathfrak{uc}}$ to the target states described in α_i.
4. Add copy and successor transitions to connect $\mathcal{A}^r_{\mathfrak{uc}}$ to $\mathcal{A}^{r+1}_{\mathfrak{uc}}$ for $1 \leq r < m$.
5. Constrain those transitions to require a valuation of $x_{\mathtt{clock}}$ within I_r.

6. Move outgoing transitions of the timelines original forks at the execution of p_{l-1} to the respective copies in $\mathcal{A}^m_{\text{uc}}$ and also adding successor transitions.

   We note that if a run takes those successor transitions to reach p_l, then the resulting trace has to put the respective entry of the platform action c at the time point t_i after the entry of p_l, so $(p_l, t_i), (c, t_i) \in \delta$. While from a practical point of view the sequence $(c, t_i), (p_l, t_i)$ expresses the same behavior, namely the concurrent start of c and p_l, they are different in the semantics of *t-ESG* as there is no notion of true concurrency. In particular, the sequence $(c, t_l), (p_l, t_l)$ could violate γ_{uc}, if c causes a state change out of the set specified in α_m, such that $\mathit{state}(\mathcal{M})$ as $\alpha_m \mathbf{U}_{I_m}$ does not hold at (c, t_l).

7. Restrict the transitions from $\mathcal{A}^m_{\texttt{sat}}$ to the forks at p_l such that valuations of $x_{\texttt{clock}}$ have to be within I_m.

Now we are almost done with the formal description of all constraints, the only one missing being $\texttt{occ}(\beta, \alpha)$, the constraint without any trace operator. However, it only requires a single step to encode that states described by α should be reached upon the start of actions specified in β: Restricting all incoming transitions of the copies at the activation p_k suffices.

Algorithm 7 ENCODE(γ_{uc}) at occurrence p_k followed by p_l

Require: $k, x_{\texttt{clock}}, \mathcal{A}_{\texttt{enc}}, \langle L^r_{\texttt{target}}, I_r\rangle, 1 \leq r \leq m, I_r = \langle v_r, w_r\rangle, P_b$
Ensure: $\mathcal{A}'_{\texttt{enc}}$ ▷ Encoding of γ_{uc} activation at p_k followed by p_l into $\mathcal{A}_{\texttt{enc}}$

$I_{\texttt{acc}} \leftarrow I_1$
for all $r \in \{2, \ldots, m\}$ **do**
  $I_{\texttt{acc}} \leftarrow I_{\texttt{acc}} + I_r$
  $\langle i_r, j_r\rangle \leftarrow$ CALCULATECONTEXT$(k, I_{\texttt{acc}}, P_b)$
  $\mathcal{A}^r_{\text{uc}} \leftarrow$ COPY(TIMELINES$(\mathcal{A}_{\texttt{enc}}, i_r, j_r), L^r_{\texttt{target}})$
end for
$\langle i_1, j_1\rangle \leftarrow$ CALCULATECONTEXT(k, I_1, P_b)
$\mathcal{A}^1_{\text{uc}} \leftarrow$ TIMELINES$(\mathcal{A}_{\texttt{enc}}, k, l-1)$
$\mathcal{A}'_{\texttt{enc}} \leftarrow \mathcal{A}_{\texttt{enc}} \cup \bigcup_{1<r\leq m} \mathcal{A}^r_{\text{uc}}$
ADDTOINCTRANS$(\mathcal{A}'_{\texttt{enc}}, k, \top, \{x_{\texttt{clock}}\})$
ADDTOINCTRANS$(\mathcal{A}'_{\texttt{enc}}, l,$ SATLB$(x_{\texttt{clock}}, I_m), \emptyset)$
for all $r \in \{1, \ldots, m-1\}$ **do**
  ADDINVARIANTS$(\mathcal{A}'_{\texttt{enc}}, \mathcal{A}^r_{\text{uc}},$ SATUB$(x_{\texttt{clock}}, I_r))$
  ADDCOPYTRANS$(\mathcal{A}'_{\texttt{enc}}, \mathcal{A}^r_{\text{uc}}, \mathcal{A}^{r+1}_{\text{uc}},$ SATLB$(x_{\texttt{clock}}, I_r), \{x_{\texttt{clock}}\}, L^{r+1}_{\texttt{target}})$
  ADDSUCCTRANS$(\mathcal{A}'_{\texttt{enc}}, \mathcal{A}^r_{\text{uc}}, \mathcal{A}^{r+1}_{\text{uc}},$ SATLB$(x_{\texttt{clock}}, I_r), \{x_{\texttt{clock}}\}, L^{r+1}_{\texttt{target}})$
end for
ADDSUCCTRANS$(\mathcal{A}'_{\texttt{enc}}, \mathcal{A}^m_{\text{uc}},$ TIMELINES$(\mathcal{A}'_{\texttt{enc}}, l, l),$ SATLB$(x_{\texttt{clock}}, I_m), \emptyset)$
SHIFTOUTGOINGTRANS$(\mathcal{A}'_{\texttt{enc}}, l-1, \mathcal{A}^1_{\text{uc}}, \mathcal{A}^m_{\text{uc}})$
RESTRICTSTATES$(\mathcal{A}'_{\texttt{enc}}, \mathcal{A}^1_{\text{uc}}, L^1_{\texttt{target}})$
ADDINVARIANTS$(\mathcal{A}'_{\texttt{enc}}, \mathcal{A}^m_{\text{uc}},$ SATUB$(x_{\texttt{clock}}, I_m))$
return $\mathcal{A}'_{\texttt{enc}}$

4.1.8 Merging Encodings of Different Models

The procedures so far only deal with a single platform component at a time, in the following we will address the general case of multiple components $\mathcal{M}_1, \ldots, \mathcal{M}_n$ that all have to be operated.

Direct Merge of Individual Transformed Plans As a starting point, let us consider what happens when we treat the control of the different $\mathcal{M}_i$ as separate problems, e.g., create n transformed plans $\delta_1, \ldots, \delta_n$ that have to be unified in the end. This approach works well, if all plans agree on the start times of high-level actions: As described in Section 3.3 the different platform actions do not interfere with each other and since the different plans agree on the domain action start times we may simply unify the plans. This can be achieved by sorting the entries across $\delta_1, \ldots, \delta_n$ based on the grounded execution times and then removing the duplicate entries for domain actions. Such a combined plan respects the constraints that every δ_i individually satisfies towards the operation of $\mathcal{M}_i$. However, issues arise once different plans determine different timings for high-level actions.

As an example let us return to the perception unit $\mathcal{M}_{\texttt{vis}}$ of Figure 2.1 to operate a camera with the same constraints as presented in Section 4.1.1. Let another component $\mathcal{M}_{\texttt{calib}}$ be responsible for calibrating the gripper after every every 30 seconds that the gripper is moving. For simplicity we assume the gripper to be moving for precisely the whole duration of a `pick` or `put` action that each take exactly 10 seconds to execute. In other words, after every third grasping task, the axis have to be re-calibrated, while the perception may be operational for up to three consecutive executions of grasping actions as well, given that the perception can be in state running for at most 30 seconds. Let the re-calibration procedure take 15 seconds and now consider the following high-level plan consisting of 6 consecutive grasping tasks modeled through durative actions with an upper bound on end actions of 20 seconds to allow platform interactions in between domain actions and an extra action `start` to allow platform control before the first action:

$$\begin{aligned}\texttt{start}(),&\texttt{start\_pick}(\vec{a}), \texttt{end\_pick}(\vec{a}), \texttt{start\_put}(\vec{a}), \texttt{end\_put}(\vec{a}),\\ &\texttt{start\_pick}(\vec{b}), \texttt{end\_pick}(\vec{b}), \texttt{start\_put}(\vec{b}), \texttt{end\_put}(\vec{b}),\\ &\texttt{start\_pick}(\vec{c}), \texttt{end\_pick}(\vec{c}), \texttt{start\_put}(\vec{c}), \texttt{end\_put}(\vec{c})\end{aligned}$$

A transformation to incorporate $\mathcal{M}_{\texttt{vis}}$ could turn on the perception at time 0, have it running at 2, proceed with the first three grasping tasks without delay, restart the perception at 32 right after the second put action is finished, have it running again by 34 to proceed with the remaining three grasping tasks leading to a total time of 64 seconds.

Considering $\mathcal{M}_{\texttt{calib}}$ instead, a transformed plan may start the first three grasping tasks right away, re-calibrate the gripper at 30 during the second `end_pick` and then continue the remaining domain actions at 45 to a total of 75 seconds.

A combination of those two plans is not possible without further adaptions, considering that during the time that the gripper calibrates, the camera continues to run unless it is explicitly turned off. The underlying problem is caused by the possibility to control the start of high-level actions that both plan transformations utilize differently causing incompatibilities.

Hierachical Ordering of Encodings One solution to fix the problem of disagreeing domain action timings could be to distinguish between components that do not need any control over high-level timings because they run strictly in parallel to any domain task and those actions that may interfere with the high-level timings. Then one could try to first obtain an encoding of the latter components that dictate the timings of domain actions when transforming the plan with respect to the former type of low-level specifics in a second step.

Combining Different Encodings This gives no solution in the above scenario, where multiple components have to influence the domain action timings. One approach is to merge those components together via a product automata construction [4] and then encode their respective constraints together into one automaton. We follow this idea, but instead of merging the initial components we first construct the encoding automata $\mathcal{A}^{\texttt{vis}}_{\texttt{enc}}$ and $\mathcal{A}^{\texttt{calib}}_{\texttt{enc}}$ separately and then merging them together afterwards.

The approach of merging late has two benefits: Firstly, it naturally produces the separate encodings as intermediate step, which may be used to validate that there are individual solutions to the constraints of the different models, e.g., by checking if the designated state `fin` is reachable in each of the encodings, which at least ensures that there are no constraints formulated for any component that form a contradiction already. Secondly, while the resulting automaton is identical, we believe it is beneficial to combine the encodings instead of the platform models to avoid unnecessary memory consumption, e.g., caused by first encoding constraints that introduce many possibilities and then incorporating constraints that restrict the available states.

We proceed to sketch the procedure of merging two encodings $\mathcal{A}^1_{\texttt{enc}}$ and $\mathcal{A}^2_{\texttt{enc}}$ together. In case of more than two platforms the given algorithm can be re-applied to merge the encodings one-by-one.

As a first step we note that during the execution of each individual domain action p_i the different platforms are operating completely independent from each other. Therefore, we can simply take the sub-automata $\mathcal{A}^1_{\texttt{enc}}|_{p_i}$ and $\mathcal{A}^2_{\texttt{enc}}|_{p_i}$ at each p_i, merge them by replacing each state of $\mathcal{A}^1_{\texttt{enc}}|_{p_i}$ by a copy of the automaton $\mathcal{A}^2_{\texttt{enc}}|_{p_i}$ and then connect the copies via copy transitions annotated according to the transitions within $\mathcal{A}^1_{\texttt{enc}}|_{p_i}$. This essentially corresponds to the technique of merging together platform and plan automaton in Section 4.1.1. The states of the resulting automata $\mathcal{A}^{1,2}_{\texttt{enc}}|_{p_i}$ are effectively tuples $\langle l^1_{\texttt{enc}}, l^2_{\texttt{enc}}\rangle$ with one state from $\mathcal{A}^1_{\texttt{enc}}$ and one

from $\mathcal{A}^2_{\texttt{enc}}$, each. The remaining task is to connect the sub-automata at p_i with the ones at p_{i+1} (or with `fin` in the case of $i = n$). Given $\langle l^1_{\texttt{enc}}, l^2_{\texttt{enc}}\rangle \in \mathcal{A}^{1,2}_{\texttt{enc}}|_{p_i}$ and $\langle \hat{l}^1_{\texttt{enc}}, \hat{l}^2_{\texttt{enc}}\rangle \in \mathcal{A}^{1,2}_{\texttt{enc}}|_{p_{i+1}}$ a transition $\langle l^1_{\texttt{enc}}, l^2_{\texttt{enc}}\rangle \xrightarrow{g\wedge\hat{g}, a+\hat{a}, r\cup\hat{r}} \langle \hat{l}^1_{\texttt{enc}}, \hat{l}^2_{\texttt{enc}}\rangle$ is added, only if $l^1_{\texttt{enc}} \xrightarrow{g,a,r} \hat{l}^1_{\texttt{enc}}$ is a transition in $\mathcal{A}^1_{\texttt{enc}}$ and $l^2_{\texttt{enc}} \xrightarrow{\hat{g},\hat{a},\hat{r}} \hat{l}^2_{\texttt{enc}}$ is contained in$\mathcal{A}^2_{\texttt{enc}}$, modeling that an execution of a domain action is only started, if the constraints of both platforms allow it.

4.2 Modular Encoding Using Communication Between Automata

The encoding procedure presented above resembles our initial approach to tackle the problem of unifying the concerns from different entities that all use different formalisms. Initial benchmarks were promising, hence we followed the track further. However, the practical problems arising from the handling of huge automata as required from the presented direct encoding are manifold: Complex datastructures are necessary to keep track of all the created copies, each created state has to get a unique name that has to carry all necessary information, leading to resulting automata that are hard to interpret for the human eye and contribute to the generally difficult task of debugging and modifying the resulting system. The initial draft was essentially a brute-force unfolding of all the possible options to transform a plan.

Driven by the preliminary results and discouraged from the unsophisticated nature of the approach to generate encodings, we experimented with extensions of the classical timed automata formalism to help with our goal towards a plan transformation that is both fast and elegant, such that handling and implementation becomes less cumbersome. Given that our starting point consists of three separate entities, namely the high-level plan, the automata for the low-level specifics and the connective constraints to specify their relations, we tried to keep those concerns separated and establish the necessary connections with the help of synchronization between different automata.

The first entity, the platform model, is already given as timed automata $\mathcal{A}_{\mathcal{M}}$, a high-level plan can be represented as timed automaton $\mathcal{A}_P$ as well, using the same construction as in Section 4.1. We proceed to also construct automata $\mathcal{A}_\gamma$ for the constraints, and then utilize synchronization channels as provided by `UPPAAL` to establish links between $\mathcal{A}_{\mathcal{M}}$ and $\mathcal{P}$ through $\mathcal{A}_\gamma$. The basic idea for this is inspired by the observation, that the operational patterns that a constraint γ imposes on a component $\mathcal{M}$ can be represented using a construction with only few copies of $\mathcal{A}_{\mathcal{M}}$, such as the ones from Figure 4.2. In fact, let us revisit the previous example

of

$$\gamma_{\texttt{prep}} := \texttt{occ}(Occ(\texttt{pick}(o,p)) \vee Occ(\texttt{put}(o,p)), \mathbf{H}_{(0,2]}(state(\mathcal{M}_{\texttt{vis}}) = \texttt{warm-up} \vee state(\mathcal{M}_{\texttt{vis}}) = \texttt{running}))$$

to showcase the construction of an automaton $\mathcal{A}_{\texttt{prep}}$ that just encodes the behavior of remaining in the target states of $\gamma_{\texttt{prep}}$ at some point and that we may link to $\mathcal{A}_P$ by means of synchronization afterwards.

Encoding Platform Behavior Separately Remembering the procedure from the direct encoding approach, the basic idea to tackle $\gamma_{\texttt{prep}}$ was to differentiate between four different situations relative to an activation at a domain action a_i within a high-level plan:

(1) Initially, $\mathcal{M}$ may be controlled freely.

(2) At some point before a_i, encoded through a non-deterministic choice of a transition to another copy, the right time in advance of a_i is assumed. In case of $\gamma_{\texttt{prep}}$ this time is exactly two seconds before a_i. Thereby, a clock is reset that counts the time towards the start of a_i and the platform is restricted to the target states specified by γ.

(3) After $\gamma_{\texttt{prep}}$ is satisfied and before a_i is started, the component may be used freely again. In this example the free usage merely equates to changing states within $\mathcal{A}_{\mathcal{M}}$ without any time elapsing.

(4) As soon as a_i is started, free control is established again. In fact, we effectively end up in (1) again, waiting for the next activation at a later plan action.

Using the idea of Figure 4.2b and remembering the usage of a clock to non-deterministically guess the right time in advance of a_i we can come up with the automaton in 4.13 that conveys the idea of the above four situations: $\mathcal{A}_{\texttt{idle}}$ models the unconstrained control over $\mathcal{M}$ while waiting for the next activation. Once the time has come, $\mathcal{A}_{\texttt{active}}$ restricts $\mathcal{M}$ to the target states of $\gamma_{\texttt{prep}}$ for two seconds, then $\gamma_{\texttt{sat}}$ deals with the remaining control before a_i starts, which should allow a move to $\mathcal{A}_{\texttt{idle}}$ again. Those transitions back are annotated with a synchronization label. Let us summarize the basic concepts of synchronization as provided by UPPAAL:
Multiple automata together form a system. If one automaton *emits* a broadcast or direct message via a *channel* by taking a transition with an attached emitting channel, then other automata in the system that could take a transition labeled as *receiver* of the broadcast are required to do so. Hence multiple automata take transitions at the exact same time via a defined master-slave relation. Broadcasts can always emitted, even if no other automaton receives them, binary channels are required to be received.

Back towards our example, $\mathcal{A}_P$ may emit a broadcast upon the start of grasping actions, such that $\mathcal{A}_{\texttt{prep}}$ (and possibly other constraints that require knowledge about grasping actions) may receive to determine the timing to move from the copy $\mathcal{A}_{\texttt{sat}}$ to $\mathcal{A}_{\texttt{idle}}$. However, there is a major flaw with the depicted automaton:

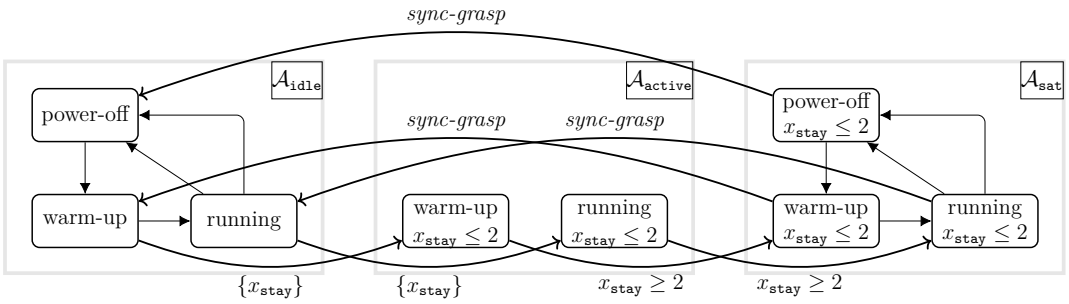


Figure 4.13: Encoding $\gamma_{\texttt{prep}}$ as separate automaton.

There is no requirement to ever leave $\mathcal{A}_{\texttt{idle}}$. As emitting broadcasts do not have to be received, $\mathcal{A}_{\texttt{prep}}$ miss the broadcast by stayin in $\mathcal{A}_{\texttt{idle}}$. Binary channels are not an option as generally other constraint encodings may require the receiving of the message from $\mathcal{A}_P$ as well. We can circumvent this issue by introducing a trap state that models the failure of any run, where $\mathcal{A}_{\texttt{prep}}$ misses the receive of a grasping start sent from $\mathcal{A}_P$. All states from $\mathcal{A}_{\texttt{idle}}$ and $\mathcal{A}_{\texttt{active}}$ are connected to said trap via transitions that are taken once $\mathcal{A}_P$ emits the broadcast. In other words, $\mathcal{A}_{\texttt{prep}}$ has to avoid the move into the trap, which is only possible by actually being in the copy $\mathcal{A}_{\texttt{sat}}$ at the right time.

Synchronized Platform Behavior Now that we get a grasp on the connection between the plan automaton and separately modeled constraint automata, we may tackle the next problem, which arises, if multiple constraints have to be encoded and they are modeled as different automata $\mathcal{A}_i, i > 1$. Then they all have to agree on the state of $\mathcal{M}$ at any given time, which we again may achieve via broadcasts:

Communication in `UPPAAL` is one-sided, flowing from sender to the receivers, hence we essentially have to model one master, that controls $\mathcal{M}$, while all automata that model concrete behavior of $\mathcal{M}$, e.g., the automaton $\mathcal{A}_{\texttt{prep}}$, mirror the moves of that master. We can simply take a single copy of $\mathcal{A}_{\mathcal{M}}$ that emits broadcasts on every single transition as such a master, but the possibility to miss emitted broadcasts has to be handled in the receiving automata yet again, e.g., while $\mathcal{A}_{\texttt{prep}}$ is in the copy $\mathcal{A}_{\texttt{active}}$ it could miss an emitted broadcast to move to `power-off`. The trap state can again be used for that matter. We depict the resulting interactions between a master automaton $\mathcal{A}_{\texttt{master}}$, $\mathcal{A}_{\texttt{prep}}$ and a plan automaton $\mathcal{A}_P$ schematically in Figure 4.14.

The Induced Model Checking Problem To obtain an executable plan from such a system of automata, we can query for paths that reach `fin` in $\mathcal{A}_P$, and

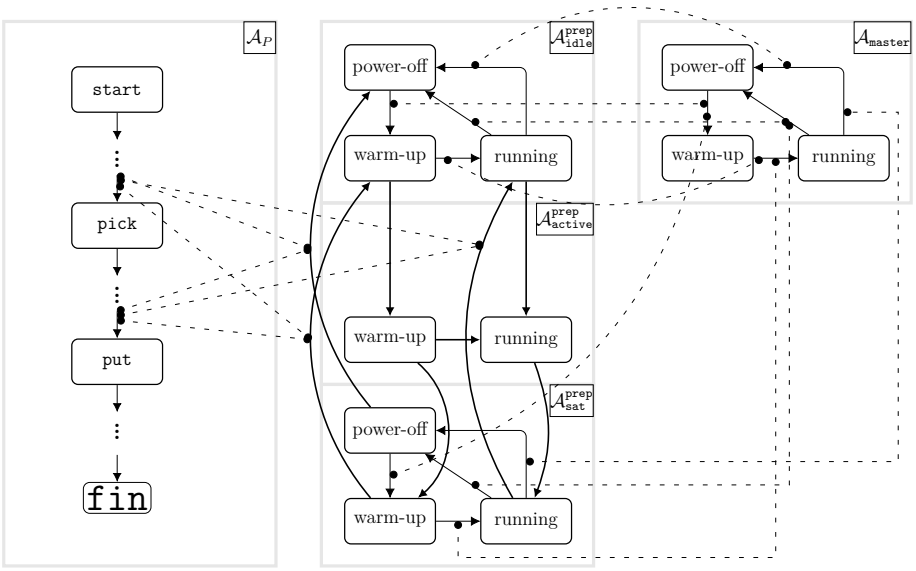


Figure 4.14: $\mathcal{A}_{\texttt{prep}}, \mathcal{A}_P$ and $\mathcal{A}_{\texttt{master}}$ as separate entities connected via communication channels (dashed lines). The trap state is not depicted.

where no constraint automaton ends up in a trap states. Preliminary tests yielded surprisingly bad results, on instances with only five synchronized automata and a plan of length 10 it already took over a minute in order to come up with a solution. We at first assumed the cause to be the used tool, as the usage of broadcast channels does not really fits our needs. However, similar tests using `Kronos`, a tool that models blocking communication, which is exactly what we need, were not promising either. We assume the real underlying problem to be the lack of explicitly encoded information, such as the order in which constraints have to be satisfied and the temporal relations between multiple activations of a constraint that may be deducible from the constraints between domain actions. While the approach presented here would be significantly easier to implement and also provides an elegant and compact encoding of the plan transformation, it ultimately is not suited to tackle the task as the objectives and constraints are obfuscated through the implicit dependencies of the different automata across the system.

5 Synthesizing Executable Plans

Returning to our overall objective to obtain a transformed plan through an solving a model checking problem of timed automata, we are now equipped with the knowledge to actually compose a reachability task that entails the possible transformed plans. However, we have yet to address the question of how to actually synthesize solution plans from a given encoding of platform model constraints $\mathcal{A}_{\mathtt{enc}}$ as described in 4.1. The reachability task to find a path to the designated state `fin` corresponds to the problem of determining an executable plan that respects the platform specifics. The next step is to utilize a model checking tool, here we side with `UPPAAL`, in order to generate such a solution.

`UPPAAL` can answer successful reeachability queries by returning a symbolic trace through the automata system, where each symbolic state is given through a minimal difference constraint system from which the DBM of the symbolic state may be computed. Our prototype implementation invokes `UPPAAL`'s command line tool `verifyta` that can produce traces that are both pre- and post-stable (see Definition 2.1.5 and Definition 2.1.6). Given such a trace $\mathcal{T} = \langle l_1, D_1\rangle \overset{g_1,a_1,r_1}{\rightsquigarrow} \langle l_2, D_2\rangle \overset{g_2,a_2,r_2}{\rightsquigarrow} \dots \overset{g_N,a_N,r_N}{\rightsquigarrow} \langle l_N, D_N\rangle$ we have to extract one of the possibly infinitely many concrete traces that are described symbolically by $\mathcal{T}$ and that each represent the same transformed plan (given through the action labels on $\mathcal{T}$) with different execution times that all satisfy the encoded constraints. In general, generating concrete traces from a symbolic trace $\mathcal{T}$ of an automaton $\mathcal{A}$ equates to determining the time points $t_1, \dots, t_N$, at which each transition within $\mathcal{T}$ is taken, such that, given the initial clock assignment μ_0 where all clocks are 0, the path

$$\langle l_1, \mu_0\rangle \xrightarrow{t_1} \langle l_1, \mu_0 + d\rangle \xrightarrow{a_1} \langle, l_2, \mu_1\rangle \xrightarrow{t_2} \langle l_2, \mu_1 + t_1\rangle \xrightarrow{a_2} \dots \xrightarrow{t_N} \langle l_N, \mu_N\rangle$$

is contained in $\mathfrak{T}(\mathcal{A})$, the semantic transition system of $\mathcal{A}$.

Determining a Concrete Trace We are particularly interested in the fastest possible run through $\mathcal{A}_{\mathtt{enc}}$ to obtain a time optimal plan with respect to the platform constraint. Such a run can be obtained by instructing `verifyta` to produce the fastest symbolic trace reaching `fin`, which in turn also contains the fastest concrete trace. In general there is no unique fastest concrete trace within given symbolic trace, e.g., if a platform action has to be invoked somewhere during some high-level action a without further constraints, then any timing on the platform action within the execution of a may be valid to form a concrete trace without interfering with the total time elapsing. We consider the concrete trace with the lowest overall elapsing time that triggers all actions as early as possible to be a canonical representation of a fastest trace to avoid ambiguity.

However, it is not always possible to actually determine optimal concrete timings for each transition of a given symbolic trace $\mathcal{T}$, as we illustrate by considering the

example in Figure 5.1. The fastest concrete trace reaching `fin` would transition to l_2 at 15, and then move to l_3 past 20. However, it is impossible to determine a concrete optimal timing as any valuation of x within $(5, 10)$ suffices for a valid trace and since clocks carry values from the domain $\mathbb{R}$ we can only provide arbitrary close approximations for the fastest trace. While we could argue that real-world executions act upon a minimal observable time delta, which could be used to give the closest approximation that is practically feasible, it does not appear to be reasonable that the actual commands that invoke the activation are controllable up to the smallest measurable precision.

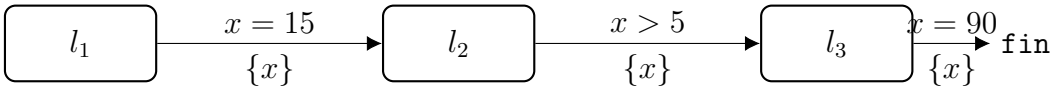


Figure 5.1: Automaton that induces no concrete fastest run towards `fin`.

Returning to our application, we believe that in cases where due to a strict lower bound on the fastest possible execution time no best solution can be given, the decision to find the exact grounding should be made within the high-level execution to start the respective action at the fastest time that is practically feasible, instead of arbitrarily choosing some time within the plan transformation procedure. We extend this line of thought and aim to always provide a range instead of a fixed grounded value for each execution timing, such that the high-level framework may be operated with some robustness towards unforeseen events that prolong or delay certain execution times.

Re-Grounding Execution Times In order to grant an executor the power to choose concrete timings, another issue has to be tackled first: The possible execution time for the i-th transition of a symbolic trace $\mathcal{T}$ of length $N \geq i$ may depend on the concrete execution times of all previously taken transitions.

Let us look at the automaton in Figure 5.2 and consider the possible traces reaching `fin`. Assuming l_3 is reached without any time elapsing, then the executor has to wait $[2, 3]$ seconds before executing a_3 and therefore reaching l_4. If on the other hand some delay of two seconds happens at l_2, then the time to wait in l_3 changes to $[0, 1]$ seconds in order to still reach `fin`. As a consequence, we propose to

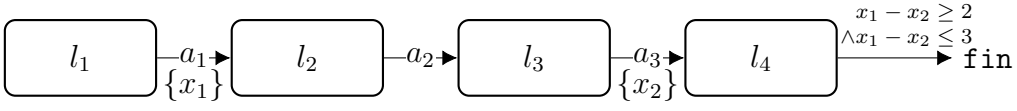


Figure 5.2: Automaton where the delays influence later possible wait times.

take the earliest feasible execution time of each transition in a symbolic trace $\mathcal{T}$ as lower bound and provide the biggest allowed delay assuming that all previous transitions were taken at their respective lower bounds as upper bound. Then,

whenever an action is executed later, either due to the provided lower bound being strict or due to some other circumstances that result in some delay, the remaining execution intervals have to be recalculated. We proceed to give details on the procedures to calculate execution start intervals and to recalculate those intervals upon encountering a delay.

5.1 Calculating Execution Start Intervals

Given a symbolic trace $\mathcal{T}$ of length N regarding a run through an automaton $\mathcal{A}$ with clocks $\mathcal{C}$, the calculation of start intervals evolves around the idea to propagate the lower bound on the current execution time while simultaneously tracking the values of clocks within $\mathcal{C}$ through the possible executions of $\mathcal{T}$.

To that end, we utilize an extra clock $\mathcal{G}$ that is added to the automata system and is never reset. Therefore, it always refers to the current execution time and we can retrieve all possible values within a symbolic state $\langle l_i, D_i \rangle$ of $\mathcal{T}$ D_i by checking the difference bounds of $\mathcal{G}$ and $\texttt{cl}_0$ in the closed form $\texttt{cf}(D_i)$ of D_i (see Definition 2.1.7). In particular, the entry corresponding to $\mathcal{G} - \texttt{cl}_0$ corresponds to the upper bound while the entry of $\texttt{cl}_0 - \mathcal{G}$ yields the negated lower bound.

Bøgsted Poulsen and van Vliet [24] show that it is indeed enough to look at the lower bounds of $\mathcal{G}$ along with the guards on the transitions that are taken within $\mathcal{T}$ to obtain a representation of the fastest concrete trace within a given post-stable trace $\mathcal{T}$. Since in our case traces are also pre-stable we may only consider the DBMs of the symbolic states without including the constraints on clock transitions. The following theorem gives an intuition for this.

Theorem 5.1.1. *Let $\langle_i l_i, u_i\rangle_i, 0 \leq i \leq N$ be the lower bound on the special clock in a forward and backward stable trace where no symbolic state imposes a strict lower bound on $\mathcal{G}$, then $\hat{l}_i := l_i, 0 \leq i \leq N$ implies a concrete trace $\tau(\hat{l}_i)$.*

Proof. By induction, claim: In step n a concrete state within D_n is reached.

Anchor $n = 0$: initially all clocks are 0. Therefore, it is a concrete state within D_0.

Step $n \mapsto n+1$: Assume that $\hat{l}_{n+1}$ is not contained in D_{n+1}. By induction $\tau(\hat{l}_n)$ is contained in D_n. Since $\mathcal{T}$ is post-stable there has to be a delay $d > l_{n+1} - l_n$ that reaches a state in in D_{n+1} from $\tau(\hat{l}_n)$ and the reason why delaying by $l_{n+1} - l_n$ is not working as delay must be due to some clock constraint enforcing a lower bound on a clock values. Other type of clock constraints (difference constraints and upper bounds) cannot be satisfied by simply waiting longer. Note that we assumed our guards do not contain disjunctions of clock constraints in Section 2.1 which reflects the restrictions of UPPAAL, so indeed a single lower bound constraint γ may be isolated as reason why l_{n+1} does not induce a symbolic state in D_{n+1}.

Since $\hat{l}_{n+1}$ is a possible clock value in D_{n+1} and $\mathcal{T}$ is backwards stable it follows that there has to be some concrete trace $o_i, 0 \leq i \leq n$ such that the special clock has value $\hat{l}_{n+1}$ in o_{n+1} and that is different from $(\tau(\hat{l}_i))_{0 \leq i \leq n}$. Since $(\tau(\hat{l}_i))_{0 \leq i \leq n}$ delayed only when absolutely necessary (as late and short as possible) up to $\tau(\hat{l}_n)$ all clocks were reset as late as possible. It follows that all clocks in $\tau(\hat{l}_n)$ have a value that is smaller or equal to the value in the predecessor state of o_{n+1}. This is contradicting to the concrete trace of $\tau(l_n)|^d$ not satisfying γ while the predecessor state of o_{n+1} did satisfy γ despite having a clock value on the mentioned clock in γ that is not bigger than the respective value in $\tau(l_n)|^d$.

So by contradiction $\hat{l}_{n+1}$ is contained in D_{n+1} which concludes the induction. □

Similar to [24] one can argue that in case there is indeed a strict lower bound on $\mathcal{G}$ a sufficiently small ϵ can be picked to obtain a concrete trace. This is not relevant for our use case as the concrete timing is determined during plan execution and therefore it suffices to return the strict bound instead.

The pseudo code to obtain execution start intervals is shown in Algorithm 8. For each transition of the symbolic trace (line 6) one can compute the lower bound on $\mathcal{G}$ in the destination state to obtain the required delay from the source state (line 7). This, together with the resets on the given transition, is used to simulate the progression of each clock (lines 8-13). With those concrete values we can update the destination state DBM to include lower bounds on each clock and calculate the maximum delay based on the smallest interval length that any clock value is restrained to (lines 14-18).

5.2 Recalculating Action Start Times

Algorithm 8 bases its calculation on the assumption that actions are started as early as possible (line 9, line 16). Whenever this is not the case during plan execution, we have to recalculate the remaining start intervals. For this we can simply treat the trace as timed automaton and add lower bound guards of $\mathcal{G}$ to the already performed transitions encoding the time that they took (including the occurred delay). Checking this automaton for a path that reaches the last trace state we obtain a fresh pre- and post-stable trace and apply Algorithm 8 again to get updated timings.

Algorithm 8 Calculate Execution Start Intervals

Require: $\mathcal{T} = D_0 \rightarrow_1 D_1 \rightarrow_2 \ldots \rightarrow_n D_n$, clocks ▷ D_i are DBMs , clocks is a set of all clock names

Ensure: $L_0, U_0, \ldots, L_n, U_n$ ▷ lower bounds to visit symbolic states and upper bounds on the duration of each visit

1: **for all** c in clocks **do**
2: $cl\_val[c] \leftarrow (0, \leq)$
3: **end for**
4: $L_0 \leftarrow$ getLowerBound(D_0)
5: $U_0 \leftarrow$ getMaxDelay(D_0)
6: **for all** $D_{i-1} \xrightarrow{g_i, r_i}_i D_i$ in $\mathcal{T}$ **do**
7: $L_i \leftarrow$ getLowerBound(D_i) ▷ determine next $\mathcal{G}$ value
8: **for all** c in clocks **do** ▷ progress clocks
9: $cl\_val[c] \leftarrow cl\_val[c] +$ getDurationLowerBound(L_i, L_{i-1})
10: **end for**
11: **for all** c in r_i **do** ▷ reset clocks
12: $cl\_val[c] \leftarrow (0, \leq)$
13: **end for**
14: $\Phi \leftarrow TRUE$
15: **for all** c in clocks **do** ▷ create lower bound constraint for clock values
16: $\Phi \leftarrow \Phi \wedge$ toLowerBound$(c, c_val[c])$
17: **end for**
18: $U_i \leftarrow$ getMaxDelay$(D_i \wedge \Phi)$
19: **end for**

6 Evaluation

We consider tasks from the *RoboCup Logistics League* (RCLL)[61, 30], which is part of the RoboCup [49], an international competition for academia and industry. The task of the league is to produce orders that arrive online by transporting workpieces to different stationary production lines. Orders consists of different pieces, namely as a base element, zero to three rings mounted on top of the base and a cap. Different MPS stations have to perform the required steps. Therefore, the basic task of autonomous robots in that domain is to pick up intermediate products from one station and deliver them to another station. Robots from different teams play on the same field and at the same time. Core requirements for a suitable robot are

- Precise gripping
- Sensors to detect machines and align to them
- Communication with machines to instruct them, communication with a referee box (a software program that administers the game, see [62])
- Navigation and localization in a dynamic environment

6.1 Benchmark Domain

We apply the plan transformation to randomly generated plans that represent possible production sequences of orders in a simplified RCLL setting, that is summarized in the following:

A basic production sequence is dependent on the number of rings and modeled through: If zero rings are required then a fixed sequence is generated via:

1. Preparing one of the two *cap stations* (randomly picked) by picking material from its' shelf and feeding it into the machine. We call this a *buffer step*.

$$\langle \texttt{goto}(s, \text{CS1}), \texttt{get-shelf}(o, \text{CS1}), \texttt{put}(o, \text{CS1}) \rangle$$

2. Retrieving the residue product from that cap station and dispose it into one of the two *ring stations* (randomly picked), also denoted as a *clean-up step*.

$$\langle \texttt{goto}(s, \text{CS1}), \texttt{pick}(o, \text{CS1}), \texttt{goto}(\text{CS1}, \text{RS2}), \texttt{pay}(o, \text{RS2}) \rangle$$

3. Transporting a base from the *base station* to the prepared cap station, such that the machine can mount the cap, also referenced as a *mount-cap step*.

$$\langle \texttt{goto}(s, \text{BS}), \texttt{pick}(o, \text{BS}), \texttt{goto}(\text{BS}, \text{CS1}), \texttt{put}(o, \text{CS1}) \rangle$$

4. Delivering the finished product to the *delivery station* in the *deliver step*.

$$\langle \texttt{goto}(s, \text{CS1}), \texttt{pick}(o, \text{CS1}), \texttt{goto}(\text{CS1}, \text{BS}), \texttt{put}(o, \text{BS}) \rangle$$

If rings have to be mounted before adding the cap, then partially ordered steps have to be applied, the sequence is determined by chance. Atomic steps are assumed to be:

1. A buffer step followed later by a clean-up step, which then enables a mount-cap step that takes a product that has all rings assembled from its current position to the buffered and cleaned-up cap station and is concluded afterwards by a deliver step.
2. *Payment steps* that require to bring a product, either from the base station or one of the cap station's shelves (randomly picked) to the cap station as payment.

$$\langle \texttt{goto}(s, \text{CS2}), \texttt{get-shelf}(o, \text{CS2}), \texttt{goto}(\text{CS2}, \text{RS1}), \texttt{pay}(o, \text{RS1}) \rangle$$

3. *Mount-ring* steps that get a workpiece from the base station (in case the first ring has to be mounted) or from its current position (if the second or third ring has to be mounted) and bring it to one of the two ring stations (predetermined for each ring by chance) only after the required payment of zero to three workpieces (the required payment is also randomized beforehand) was provided. It is possible that a product requires multiple rings from the same station in the row. Then the product has to be picked up again and freshly placed on the machine in order to trigger the next mounting step.

$$\langle \texttt{goto}(s, \text{RS1}), \texttt{pick}(o, \text{RS1}), \texttt{goto}(\text{RS1}, \text{RS2}), \texttt{put}(o, \text{RS2}) \rangle$$

This allows for multiple possible orderings of steps to assemble products, e.g., there are 33 reasonable sequences for producing an order requiring a cap from the first cap station and a ring with cost 1 from the second ring station.

The randomized plan generation provides us with the means to test our proposed plan transformation on plans of different sizes and forms and that contain the following actions:

- $\texttt{pick}(o, m)$ to pick an object o from a machine m.
- $\texttt{get-shelf}(o, m)$ to pick an object o from the shelf of a cap station m.
- $\texttt{pay}(o, m)$ to put an object o into the payment slide of a ring station m.
- $\texttt{put}(o, m)$ to put an object o into a machine m.
- $\texttt{goto}(m, m')$ to move from a machine m to m'.

We model those actions to be durative, so every action is actually split into a start and end action, we also add designated `plan-start` and `plan-end` actions as first

and last entry of each plan, which will become handy, when modeling platform constraints. For convenience we establish the following notions:

$$A^{\texttt{start}}_{\texttt{pick}}(o,m) := \{\texttt{start\_pick}(o,m), \texttt{start\_get-shelf}(o,m)\}$$
$$A^{\texttt{start}}_{\texttt{put}}(o,m) := \{\texttt{start\_put}(o,m), \texttt{start\_pay}(o,m)\}$$
$$A^{\texttt{start}}_{\texttt{grasp}}(o,m) := A^{\texttt{start}}_{\texttt{pick}}(o,m) \cup A^{\texttt{start}}_{\texttt{put}}(o,m)$$

Analogously, we define the corresponding sets for end actions.

6.1.1 High-Level Temporal Constraints

It remains to formalize the temporal relations for each of the generated plans. Given a plan $P = \langle a_1 = \texttt{plan-start}, a_2, \ldots, a_n = \texttt{plan-end}\rangle$ we define:

- $\mathfrak{rel}(1, 2, [0, 0])$, so the `plan-start` consumes no time.
- $\mathfrak{rel}(k, k+1, [15, 20])$, if $k \mod 2 = 0$ and the k-th action is $a(o,m) \in A^{\texttt{start}}_{\texttt{grasp}}$. This constrains the duration of grasping tasks to be within $[15, 20]$ as we assume strictly sequential domain actions, such that each start action is directly followed by its' respective end action.
- $\mathfrak{rel}(k, k+1, [30, 45])$, if $k \mod 2 = 0$ and the k-th action is `start_goto`.
- $\mathfrak{rel}(k, k+1, [0, 30])$, if $k \mod 2 = 1$ (the k-th action is a end action)
- $\mathfrak{abs}(1, [0, 30])$, such that the abstract plan starts to get executed at most 30 seconds after the begin of the transformed plan.
- $\mathfrak{abs}(k, I)$ for $k > 1$ can be deduced from the given relative constraints.

6.2 Platform Models

We evaluate the proposed plan transformation procedure by considering robots with a gripper consisting of x, y, z axis, a perception unit based of data from a depth camera, and communication interfaces to instruct machines as well as to report the current position as needed. The presented use cases are inspired by discussions about improvements to the reasoning system of the RoboCup team *Carologistics*.

6.2.1 Perception Unit

The perception unit $\mathcal{M}_{\texttt{perc}}$ is modeled by the automaton $\mathcal{A}_{\mathcal{M}_{\texttt{perc}}}$ depicted in Figure 6.1. It models four different concerns: Firstly, the control of the camera, which should be powered off when not needed. The second responsibility is to trigger the detection of the conveyor belts on each machine, such that the gripper can be

aligned precisely. This requires to perform an iterative procedure of point-cloud matching based in ICP, which takes between 10 to 15 seconds based on the quality of data and therefore determines the actual duration of each grasping task. Another task of $\mathcal{M}_{\texttt{perc}}$ is to upload a picture from the camera after the successful alignment to the machines. The alignment is done after the point-cloud matching is established and takes at most 5 seconds, hence pictures have to be taken afterwards. Said pictures are useful to provide training data in order to evaluate the quality of the alignment through a neural network. Lastly, a infrared sensor is attached to the gripper in order to monitor the results of grasping tasks by sensing whether objects are currently being held or not. However, since the depth camera interferes with the quality of the infrared sensor, the camera has to be turned off in order to obtain reliable results. Moreover, the camera keeps interfering with the sensor for a short period of time after it has already been turned off, as the internal hardware keeps the depth stream running for a while. To account for this, the puck sense has to wait an additional two seconds after the camera has been shut off.

The expected benefits from using a plan transformation as opposed to abstracting the hard-ware control in low-level procedures unaware of the high-level context are manifold: On the one hand it allows to make the decision to turn-off the camera based on future usage, e.g, the camera should stay powered across sequences of `pick` and `put` actions. It also admits fine-grained control over the puck sensor, by defining its usage only as need, e.g., if grasping tasks perform robustly, during movements the sensor data becomes noisy and there is a chance to lose a product while driving, a puck check could be demanded after each `goto` that follows a `pick`. Lastly, $\mathcal{M}_{\texttt{perc}}$ also demonstrates the power to model the temporal relations between execution time and uncertain low-level behavior, as we will show later.

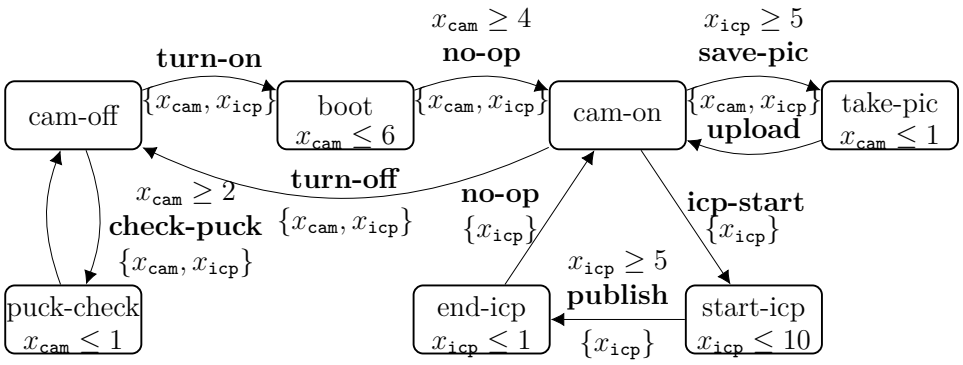


Figure 6.1: Automaton $\mathcal{A}_{\mathcal{M}_{\texttt{perc}}}$ to refine grasping actions.

6.2.2 Axis Calibration

A second model $\mathcal{M}_{\texttt{calib}}$ takes care of maintaining the precision of the axis movements that align the gripper. The precision of the axis movements degrades over time, which is modeled by a cycle of at most two axis alignments before re-calibration becomes necessary, which takes 20 seconds and consists of moving each axis to one of their endpoints. The automaton to model this hardware control is depicted in Figure 6.2.

A plan transformation may enable to effectively schedule the calibration as needed and under considerations of safety concerns, such as avoiding calibrations when a workpiece is currently being held as it induces a risk of losing the held object in the process.

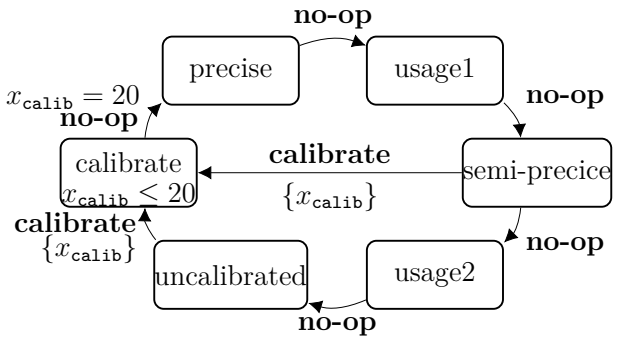


Figure 6.2: Automaton $\mathcal{A}_{\mathcal{M}_{\texttt{calib}}}$ to control axis calibration.

6.2.3 Communication Interfaces

Figure 6.3 shows the automaton $\mathcal{A}_{\mathcal{M}_{\texttt{comm}}}$ of a model $\mathcal{M}_{\texttt{comm}}$ to handle machine instructions. Under the assumption that robots only bring a workpiece to a machine, if the intended assembly step can be performed, the machine instructions may be decoupled from the domain by instructing a machine after a product has been delivered to the machine and before it is retrieved. In order to evaluate the scaling of our approach with growing number of platform models, we will consider four copies $\mathcal{A}_{\mathcal{M}_{\texttt{comm1}}}, \mathcal{A}_{\mathcal{M}_{\texttt{comm2}}}, \mathcal{A}_{\mathcal{M}_{\texttt{comm3}}}$ and $\mathcal{A}_{\mathcal{M}_{\texttt{comm4}}}$ of the presented automaton to account for communication with the two cap stations and the ring stations.

We believe that such an abstraction could significantly simplify the abstract domain, as it permits a very high-level view on machine interactions.

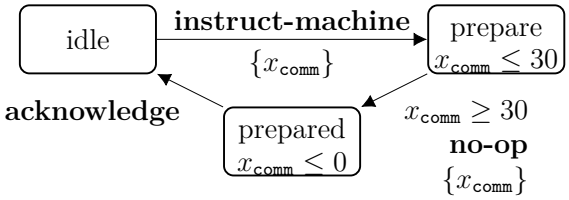


Figure 6.3: Automaton $\mathcal{A}_{\mathcal{M}_{\texttt{comm}}}$ to prepare machines.

6.3 Platform Constraints

The final prerequisites are the constraints to define the usage of platforms in the context of the given domain plans. We essentially formalize the utility of the four different models with the help of the formulas from Section 4. In order to establish a concise notion, the following abbreviations are used:

- $\phi_{\texttt{loc}} := \mathit{state}(\mathcal{M}) = \texttt{loc}$, the platform $\mathcal{M}$ can be derived from the context in which $\phi_{\texttt{loc}}$ appears.
- $\psi_{\texttt{act}}(\vec{x}) := \mathit{Occ}(\texttt{act}(\vec{x}))$..
- $\Psi^{\texttt{start}}_{\texttt{grasp}}(\vec{x}) := \bigvee_{\texttt{act} \in A^{\texttt{start}}_{\texttt{grasp}}} \mathit{Occ}(\texttt{act}(\vec{x}))$ and similarly for the other sets defined in Section 6.1.1.
- $\Phi_{\texttt{target}} := \bigvee_{\texttt{loc} \in L_{\texttt{target}}} \alpha_{\texttt{loc}}$ for a set of platform states $L_{\texttt{target}}$ and using $L_\top$ to denote all states of $\mathcal{M}$.
- $I_\top := [0, \infty)$

Let us start with the constraints for $\mathcal{M}_{\texttt{perc}}$ of Figure 6.1. Using $L_{\texttt{no-icp}} := L_\top \setminus \{\texttt{start-icp}, \texttt{end-icp}\}$ and $L_{\texttt{no-cam}} := \{\texttt{cam-off}, \texttt{puck-check}\}$, we constrain $\mathcal{M}_{\texttt{perc}}$ as follows:

$$\begin{aligned}
\gamma^1_{\texttt{perc}} :=& \mathfrak{uc}\Big[\langle(\phi_{\texttt{icp-start}}, I_\top), (\phi_{\texttt{icp-end}}, [0,0]), (\Phi_{\texttt{no-icp}}, [10,10])\rangle, \Psi^{\texttt{start}}_{\texttt{grasp}}(o,m), \\
& \qquad \Psi^{\texttt{end}}_{\texttt{grasp}}(o,m)\Big] \\
\gamma^2_{\texttt{perc}} :=& \mathfrak{uc}\Big[\langle(\Phi_\top, I_\top), (\phi_{\texttt{take-pic}}, I_\top), (\Phi_\top, I_\top)\rangle, \Psi^{\texttt{start}}_{\texttt{grasp}}(o,m), \Psi^{\texttt{end}}_{\texttt{grasp}}(o,m)\Big] \\
\gamma^3_{\texttt{perc}} :=& \mathfrak{uc}\Big[\langle(\Phi_{\texttt{no-cam}}, I_\top)\rangle, \psi_{\texttt{start_goto}}(m,m'), \psi_{\texttt{end_goto}}(m,m')\Big] \\
\gamma^4_{\texttt{perc}} :=& \mathfrak{occ}\Big[\psi_{\texttt{start_goto}}(m,m'), \phi_{\texttt{take_pic}}\Big] \\
\gamma^5_{\texttt{perc}} :=& \mathfrak{occ}\Big[\psi_{\texttt{end_goto}}(m,m'), \phi_{\texttt{take_pic}}\Big]
\end{aligned}$$

$\gamma^1_{\texttt{perc}}$ states that the ICP procedure should start directly when starting a grasping task and afterwards constraints $\mathcal{M}_{\texttt{perc}}$ to not invoke ICP again. It also implicitly specifies the run-time of the belonging high-level action: When said action starts, $\mathcal{M}_{\texttt{perc}}$ is forced to move to `end-icp` $[5, 10]$ seconds afterwards due to the modeling

of $\mathcal{A}_{\mathcal{M}_{\texttt{perc}}}$. Afterwards it permits a duration of exactly 10 seconds before the grasping action has be done. Hence this refinement connects the operation window of grasping tasks given in Section 6.1.1 with the underlying hardware-specific reason. This design was also chosen to demonstrate how careful platform designers have to be when utilizing until-chain constraints, since implications through the structure of the platform automaton and the composed chain can easily impose restrictions on the time interval between two domain actions. $\gamma^2_{\texttt{perc}}$ operates on the same active window as $\gamma^1_{\texttt{perc}}$ by demanding the perception unit to upload a picture during each grasping task. Lastly, $\gamma^3_{\texttt{perc}}, \gamma^4_{\texttt{perc}}$ and $\gamma^5_{\texttt{perc}}$ manage the control during `goto`, by requiring that the camera should be turned off and the puck sensor should be checked when `goto` starts and ends.

Towards $\mathcal{M}_{\texttt{calib}}$ we first define $L_{\texttt{use}} := \{\texttt{usage1}, \texttt{usage2}\}$, $L_{\texttt{no-use}} := L_\top \setminus L_{\texttt{use}}$ and $L_{\texttt{no-calib}} := L_\top \setminus \{\texttt{calibrate}\}$, such that we can proceed to define the following constraints:

$$\gamma^1_{\texttt{calib}} := \mathfrak{uc}\Big[\langle(\Phi_{\texttt{no-use}}, I_\top)\rangle, \psi_{\texttt{start\_goto}}(m, m'), \psi_{\texttt{end\_goto}}(m, m')\Big]$$
$$\gamma^2_{\texttt{calib}} := \mathfrak{uc}\Big[\langle(\Phi_{\texttt{no-calib}}, I_\top)\rangle, \Psi^{\texttt{end}}_{\texttt{pick}}(o, m), \psi_{\texttt{start\_put}}(o, m)\Big]$$
$$\gamma^3_{\texttt{calib}} := \mathfrak{uc}\Big[\langle(\Phi_\top, I_\top), (\phi_{\texttt{precice}}, I_\top), (\Phi_\top, [0, 0])\rangle, \Psi^{\texttt{end}}_{\texttt{pick}}(o, m), \psi_{\texttt{end\_pay}}(o, m)\Big]$$
$$\gamma^4_{\texttt{calib}} := \mathfrak{uc}\Big[\langle(\Phi_{\texttt{use}}, I_\top)\rangle, \Psi^{\texttt{start}}_{\texttt{grasp}}(o, m), \Psi^{\texttt{end}}_{\texttt{grasp}}(o, m)\Big]$$

$\gamma^1_{\texttt{calib}}$ ensures that the states that model axis usage are not visited during `goto`. Since $\mathcal{M}_{\texttt{calib}}$ does not invoke any actual platform actions upon entering those states, $\gamma_{\texttt{calib}}$ is not necessary from a modeling perspective. However, the performance of the encoding may benefit from having said constraint, because the resulting encodin removes unnecessary branching possibilities.
The next two constraints $\gamma^2_{\texttt{calib}}$ and $\gamma^3_{\texttt{calib}}$ are used to represent context-based calibration: The former states, that the axis should not be calibrated, while an object is being held that gets placed on a machines' conveyor belt. This may be beneficial, because on the one hand, axis movement through calibration may come with a small chance of a product getting dropped by accident, on the other hand it also means that there is no time consuming calibration while an important workpiece is hold, which again may also increase the risk of losing a workpiece due to external reasons. $\gamma^3_{\texttt{calib}}$ practically enforces the opposite behavior if the held workpiece should be used as payment. The grasping task to successfully pay a workpiece at a ring station is significantly different to grasping tasks on the conveyor belts at the machines. $\gamma^3_{\texttt{calib}}$ guarantees that the gripper is freshly calibrated to be as precise as possible, before attempting the payment task.
Lastly, $\gamma^4_{\texttt{calib}}$ models the progression from one usage state into the next one to actually enforce a re-calibration after two consecutive grasping tasks.

The machine instructions are modeled with the following constraints, where $\hat{m}$ denotes the machine that the respective platform model controls:

$$\gamma^1_{\texttt{comm}} := \mathfrak{uc}\Big[\langle(\phi_{\texttt{idle}} \vee \phi_{\texttt{prepare}}, I_\top), (\phi_{\texttt{prepared}}, I_\top)\rangle, \psi_{\texttt{end\_put}}(o, \hat{m}), \psi_{\texttt{start\_pick}}(o, \hat{m}) \vee \psi_{\texttt{end\_plan}}()\Big]$$

$$\gamma^2_{\texttt{comm}} := \mathfrak{uc}\Big[\langle(\phi_{\texttt{idle}}, I_\top)\rangle, \psi_{\texttt{end\_pick}}(o, \hat{m}) \vee \psi_{\texttt{strt\_plan}}(), \psi_{\texttt{end\_put}}(o', \hat{m}) \vee \psi_{\texttt{end\_plan}}()\Big]$$

While constraint $\gamma^1_{\texttt{comm}}$ ensures that $\hat{m}$ is prepared whenever necessary (before a belonging `pick` action or before the plan ends, in case an object remains in the machine until the end of the plan), $\gamma^2_{\texttt{comm}}$ states that machine preparation is not necessary as long as no product is placed int $\hat{m}$.

The definition of the above constraints concludes the description of our test domain, such that we can proceed with practical results to get an idea of the capabilities of the plan transformation procedure developed through this thesis.

6.4 Benchmarks

We argue that our developed procedure should act as a transparent pos-processing step within an execution framework, as such we consider it a necessity to transform plans in real-time without inducing significant overhead to the system.

Therefore, we focus the practical evaluation to determining the bounds both in terms of the number of decoupled platforms and the lengths of abstract plans that the presented plan transformation can realistically handle. The results we are going to show were produced using a prototype implementation of the plan transformation procedure called `taptenc` (**t**imed **a**utomata based **p**lan **t**ransformation **e**ncoding), which is available on GitHub [1]. Tests were carried out on an Intel i7-3632QM 2.2 GHz processor.

Towards the former concern, we subsequently merged together the components presented above on randomly generated plans of length 50. Merging was done by combining all encodings before the model checking step (see Section 4.1.1), so that the resulting reachability task had to consider all imposed constraints at once. Each encoding was also considered separately, results are shown in Figure 6.4, where $\mathcal{M}_{\texttt{merge1}}$ refers to the merged encoding of $\mathcal{M}_{\texttt{perc}}$ and $\mathcal{M}_{\texttt{grasp}}$. The remaining automata $\mathcal{M}_{\texttt{merge2}}$,$\mathcal{M}_{\texttt{merge3}}$ and $\mathcal{M}_{\texttt{merge4}}$ include subsequently more instances of the communication automaton. The time was measured separately for the different steps towards the resulting transformed plan: First our tool `taptenc` creates the encoding using the algorithms presented in Section 4.1, then the command line tool `verifyta` from the `UPPAAL` suite is used to create the intermediate format required to perform model checking procedures. The reachability task was carried out using the command line options `-t 2 -Y` to obtain the fastest symbolic trace with pre- and post-stability ensured. Afterwards `taptenc` was used to convert the results into a transformed plans.

[1] https://github.com/TarikViehmann/taptenc

	TIME(ms) ENCODING	TIME(ms) LOAD_MODEL	TIME(ms) REACH	TIME(ms) TRACER	TIME(ms) SYNTHESIS	NUM STATES
$\mathcal{M}_{\texttt{perc}}$	450	218	163	49	16	660
$\mathcal{M}_{\texttt{calib}}$	84	72	53	22	9	284
$\mathcal{M}_{\texttt{comm1}}$	21	25	20	9	3	62
$\mathcal{M}_{\texttt{comm2}}$	23	28	23	11	3	71
$\mathcal{M}_{\texttt{comm3}}$	22	23	21	9	3	58
$\mathcal{M}_{\texttt{comm4}}$	24	24	19	11	4	60
$\mathcal{M}_{\texttt{merge1}}$	628	1873	1324	208	38	2915
$\mathcal{M}_{\texttt{merge2}}$	958	2773	1910	295	49	3471
$\mathcal{M}_{\texttt{merge3}}$	1885	5188	3463	476	61	4952
$\mathcal{M}_{\texttt{merge4}}$	4013	8345	5249	714	95	6673
$\mathcal{M}_{\texttt{merge5}}$	15721	20496	11560	1387	314	11806

Figure 6.4: Average results of five runs on randomized plans of length 50.

We can clearly see how much overhead the naive unification of different encodings introduces to the overall performance, despite the relatively small size of most of the considered automata. However, we can also observe that the approach is feasible in principle as the merge of four components on a medium sized plans is doable in about 10 s. By utilizing further optimizations such as hierarchical merging strategies and by designing better pruning algorithms, we expect to already perform sufficiently well to be viable in real-world scenarios. Looking closer at Figure 6.4 it also appears that the size of the encoded automata is a major factor considering both the time spent by `verifyta` to compute the intermediate format and by `taptenc` to produce the encoding. Interestingly, the model checking task itself is not the limiting factor in the conducted tests, which we assume is because of the guidance that the explicit encoding gives to the underlying task by unfolding the desired solution path. Given that the approach in Section 4.2 imposed a model checking challenge that was too complex to solve, we believe that there is a fine line between the extremes we came up with so far, that leads to a better balance between the model checking procedure and the encoding size and that has yet to be found.

We also were interested in the impact that the plan length has on the transformation procedure, so we also conducted a second set of tests, where the plan length was subsequently extended to $100, 150$ and 300, based on which the encoding of each individual component, as well as $\mathcal{M}_{\texttt{merge1}}$ was computed. We ran the tests five times and considered the average results of the time it took on the encoder side versus the time spent on the encoding, the results are shown in Figure 6.5. The results look quite promising already, given that plans with 150 domain actions were no major issue. Without much surprise, increasing the plan length seems to lead to a linear growth of the encoded systems, which is also depicted in Figure 6.6.

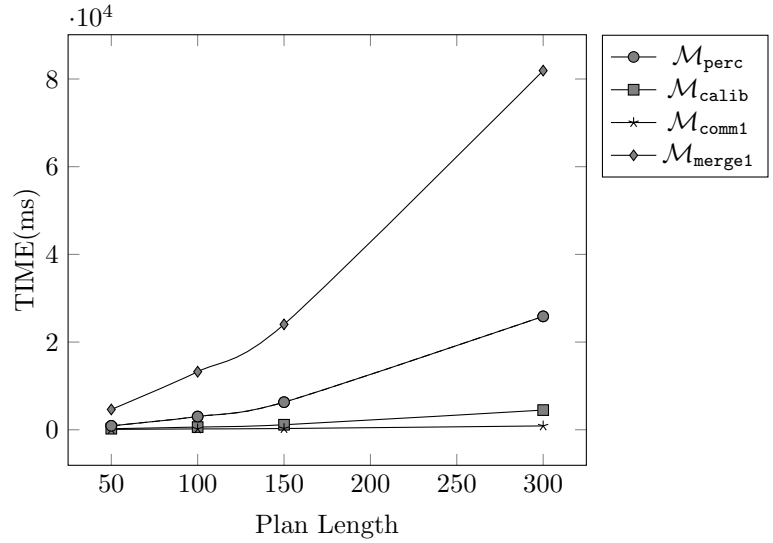


Figure 6.5: Average results of five runs on randomized plans of length $50, 100, 150$ and 300. Times of the different steps towards the resulting transformed plan are accumulated.

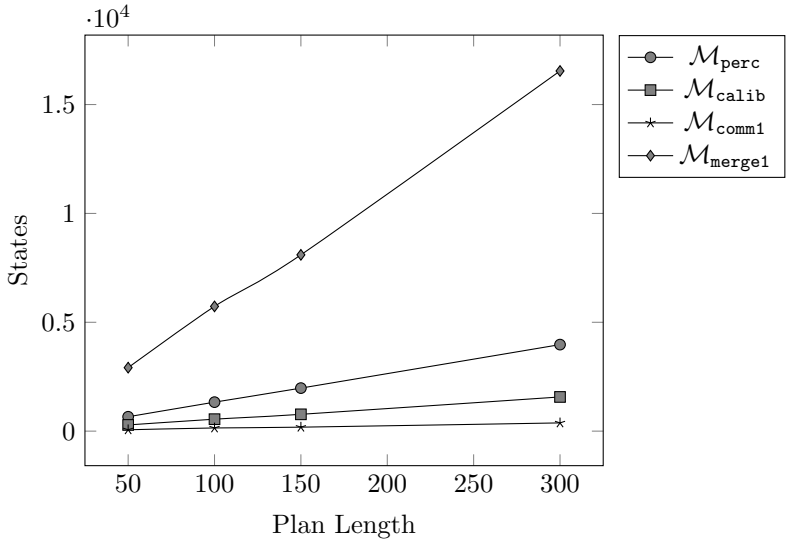


Figure 6.6: Average encoding sizes (measured by the number of states) of five runs on randomized plans of length $50, 100, 150$ and 300.

6.5 Improvements and Limitations

The showcased benchmark should serve as evidence for the diverse possibilities our developed procedure offers towards the modeling capabilities when expressing platform-specific implications on the domain of reasoning. We want to also provide more ideas to further enrich the formalism in order to cover more problems from the actual world that are currently outside of the scope of the provided formalisms.

Looking at the perception unit, we essentially modeled execution uncertainties that arise from hardware specifics through establishing a connection between grasping action durations and the runtime of ICP. However, typically those uncertainties are not controllable by any hardware-interface, but just happen through a mix of exogenous events, variances in the available data and unpredictable interference from other agents, humans and other foreign entities within the environment.

Our approach can try to deal with execution uncertainties by recalculating timings of the provided plan. However, there is no notion of uncontrollable behavior in the classical timed automata formalism. While we can specify intervals for actions with the help of the constraints of Section 3.4, those do not carry the intended semantics: In the end it is up to the model checker to determine the optimal duration for each action, which could even demand to execute an action longer than it takes to physically perform it. A more accurate depiction of uncertain durations could be acquired by utilizing the formalism of timed game automata[7], which enable to model actions via transitions that the model checker cannot control. This also requires to re-evaluate the expressive power that constraints connecting hardware specifics with domain concerns have: Facing uncontrollable actions typically equates to having a control flow that is rather reactive than proactive. Constraints that require past operation relative to occurring actions do not really make sense in light of uncontrollable high-level actions, as there generally is no way to fulfill them. Similarly, constraints such as the provided until chains have to be critically questioned as they postulate precise temporal behavior between action patterns, which resembles an uncertain time span.

A different issue we recognized when modeling low-level specifics with timed automata stems from the need to not only model the intended behavior, but to also consider the possible control patterns that the model checker might produce when platform usage is not restricted. Optimizations towards the fastest possible trace do not exclude unnecessary invocations of low-level control actions that do not impact the overall time span of the solution. It may be useful to have a notion of default behavior, such that the focus when designing platform constraints can be on the intended control rather than on the avoidance of unintended effects. One possibility is to use a notion of priced timed automata[13] to add rewards to certain states such that the model checker only leaves them to pursue a desirable objective.

7 Conclusion

This thesis developed a procedure to transform an abstract plan into an executable one while respecting platform specifics that are not part of the abstract domain. Having platform components decoupled from the high-level domain allows to adapt low-level specifics without changing the agent code as different concerns are clearly separated. The well-understood formalism of timed automata was used to represent platform components, while the connection between such automata models and the abstract domain were expressed with metric temporal constraints from a subset of t-$\mathcal{ESG}$ formulas. The two formalisms together allow to express complex temporal relations, not only between different control states of a low-level component modeled as a timed automaton, but also through logical formulas to leverage the close temporal control in the high-level domain context, while not imposing any requirements on the used domain planners. A transformation of an abstract plan into an executable one respecting the connective constraints of the platform specifics was encoded as a reachability problem on timed automata. Therefore, the developed procedure benefits from progress in the field of model checking and existing tools can be used for the computation.

The practical evaluation strongly suggests the feasibility of the presented plan transformation in real-world scenarios and gives an idea of the modeling capabilities that are entailed by the approach. We considered use-cases that could benefit from the developed procedure in the context of industrial production-line scenarios, where a core requirement is to efficiently perform precise grasping tasks. Expressive formalism to model the involved low-level specifics may greatly increase the overall performance in those domains, because complex interactions between different hardware specifics allow for various optimizations when the high-level domain context is considered. We provided models to conquer different problems that were encountered when programming real-world robotic platforms and the preliminary results turned out promising, such that an integration into robotic frameworks is subject to ongoing work.

To summarize, this thesis explored ways to model low-level specifications as timed automata in order to separate them from the high-level domain and to transform high-level plans into executable ones, by respecting the requirements of the modeled components expressed through metric temporal constraints. The problem of computing executable plans while respecting the modeled dependencies is tackled by combining the different constraints into a timed automaton and encoding the plan transformation task as a reachability problem that can be solved by available model checking tools.

Bibliography

[1] Allen, J. F. (1983). Maintaining Knowledge About Temporal Intervals. *Commun. ACM*, 26(11):832–843.

[2] Alur, R., Courcoubetis, C., and Dill, D. (1990). Model-checking for real-time systems. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425.

[3] Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. (1995). The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3 – 34.

[4] Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235.

[5] Anderson, K., Holte, R., and Schaeffer, J. (2007). Partial Pattern Databases. In Miguel, I. and Ruml, W., editors, *Abstraction, Reformulation, and Approximation*, pages 20–34, Berlin, Heidelberg. Springer Berlin Heidelberg.

[6] Asarin, E. and Maler, O. (1999). As Soon as Possible: Time Optimal Control for Timed Automata. In Vaandrager, F. W. and van Schuppen, J. H., editors, *Hybrid Systems: Computation and Control*, pages 19–30, Berlin, Heidelberg. Springer Berlin Heidelberg.

[7] Asarin, E., Maler, O., Pnueli, A., and Sifakis, J. (1998). *Controller Synthesis For Timed Automata*.

[8] BACKHOUSE, R. C. and CARRÉ, B. A. (1975). Regular Algebra Applied to Path-finding Problems. *IMA Journal of Applied Mathematics*, 15(2):161–186.

[9] Baier, C., Katoen, J., and Larsen, K. (2008). *Principles of Model Checking*. Mit Press. MIT Press.

[10] Bayless, S., Bayless, N., Hoos, H. H., and Hu, A. J. (2015). SAT Modulo Monotonic Theories. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 3702–3709. AAAI Press. event-place: Austin, Texas.

[11] Behrmann, G., David, A., and Larsen, K. G. (2004). A Tutorial on \sc Uppaal. In Bernardo, M. and Corradini, F., editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, LNCS, pages 200–236. Springer–Verlag.

[12] Behrmann, G., Fehnker, A., Hune, T., Larsen, K., Pettersson, P., Romijn, J., and Vaandrager, F. (2001). Minimum-Cost Reachability for Priced Time Automata. In Di Benedetto, M. D. and Sangiovanni-Vincentelli, A., editors, *Hybrid Systems: Computation and Control*, pages 147–161, Berlin, Heidelberg. Springer Berlin Heidelberg.

[13] Behrmann, G., Larsen, K. G., and Rasmussen, J. I. (2005). Priced Timed Automata: Algorithms and Applications. In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W.-P., editors, *Formal Methods for Components and Objects*, pages 162–182, Berlin, Heidelberg. Springer Berlin Heidelberg.

[14] Bengtsson, J. and Yi, W. (2003). On Clock Difference Constraints and Termination in Reachability Analysis of Timed Automata. In Dong, J. S. and Woodcock, J., editors, *Formal Methods and Software Engineering*, pages 491–503, Berlin, Heidelberg. Springer Berlin Heidelberg.

[15] Bengtsson, J. and Yi, W. (2004). Timed Automata: Semantics, Algorithms and Tools. In Desel, J., Reisig, W., and Rozenberg, G., editors, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg.

[16] Bogomolov, S., Donzé, A., Frehse, G., Grosu, R., Johnson, T. T., Ladan, H., Podelski, A., and Wehrle, M. (2016). Guided search for hybrid systems based on coarse-grained space abstractions. *International Journal on Software Tools for Technology Transfer*, 18(4):449–467.

[17] Bouajjani, A., Tripakis, S., and Yovine, S. (1997). On-the-fly symbolic model checking for real-time systems. *Proceedings Real-Time Systems Symposium*, pages 25–34.

[18] Bouyer, P., Cassez, F., Fleury, E., and Larsen, K. G. (2005). Optimal Strategies in Priced Timed Game Automata. In Lodaya, K. and Mahajan, M., editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, pages 148–160, Berlin, Heidelberg. Springer Berlin Heidelberg.

[19] Bouyer, P., Dufourd, C., Fleury, E., and Petit, A. (2000). Are Timed Automata Updatable? In Emerson, E. A. and Sistla, A. P., editors, *Computer Aided Verification*, pages 464–479, Berlin, Heidelberg. Springer Berlin Heidelberg.

[20] Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., and Yovine, S. (1998). Kronos: A model-checking tool for real-time systems. In Ravn, A. P. and Rischel, H., editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 298–302, Berlin, Heidelberg. Springer Berlin Heidelberg.

[21] Bérard, B. and Dufourd, C. (2000). Timed Automata and Additive Clock Constraints. In *Information Processing Letters*, pages 75–1.

[22] Bérard, B., Petit, A., Diekert, V., and Gastin, P. (1998). Characterization of the Expressive Power of Silent Transitions in Timed Automata. *Fundam. Inf.*, 36(2,3):145–182.

[23] Bérard, B. and Sierra, L. (2019). Comparing verification with HyTech, Kronos and Uppaal on the railroad crossing example.

[24] Bøgsted Poulsen, D. and van Vliet, J. (2010). *Concrete Delays for Symbolic Traces*. Computer Science, Master, Aalborg University, Department of Computer Science.

[25] Cassez, F., David, A., Fleury, E., Larsen, K. G., and Lime, D. (2005). Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In Abadi, M. and de Alfaro, L., editors, *CONCUR 2005 – Concurrency Theory*, pages 66–80, Berlin, Heidelberg. Springer Berlin Heidelberg.

[26] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M. (2000). NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425.

[27] Cimatti, A., Giunchiglia, E., Giunchiglia, F., and Traverso, P. (1997). Planning via model checking: A decision procedure for AR. In Steel, S. and Alami, R., editors, *Recent Advances in AI Planning*, pages 130–142, Berlin, Heidelberg. Springer Berlin Heidelberg.

[28] Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching time temporal logic. In Kozen, D., editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg. Springer Berlin Heidelberg.

[29] Claßen, J. and Lakemeyer, G. (2008). A Logic for Non-terminating Golog Programs. In *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning*, KR'08, pages 589–599. AAAI Press. event-place: Sydney, Australia.

[30] Coelen, V., Deppe, C., Gomaa, M., Hofmann, T., Karras, U., Niemueller, T., Rohr, A., and Ulz, T. (2019). *The RoboCup Logistics League Rulebook for 2019*. RoboCup Logistics League Technical Committee.

[31] de Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[32] De Moura, L. and Bjørner, N. (2011). Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77.

[33] Dill, D. L. (1990). Timing assumptions and verification of finite-state concurrent systems. In Sifakis, J., editor, *Automatic Verification Methods for Finite State Systems*, pages 197–212, Berlin, Heidelberg. Springer Berlin Heidelberg.

[34] Eén, N. and Sörensson, N. (2004). An extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, pages 333–336.

[35] Floyd, R. W. (1962). Algorithm 97: Shortest Path. *Commun. ACM*, 5(6):345–.

[36] Frehse, G., Le Guernic, C., 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 Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, pages 379–395, Berlin, Heidelberg. Springer Berlin Heidelberg.

[37] Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. (2007). clasp: A Conflict-Driven Answer Set Solver. In Baral, C., Brewka, G., and Schlipf, J., editors, *Logic Programming and Nonmonotonic Reasoning*, pages 260–265, Berlin, Heidelberg. Springer Berlin Heidelberg.

[38] Ghallab, M., Howe, A., Knoblock, C., Mcdermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). *PDDL—The Planning Domain Definition Language*.

[39] Giunchiglia, F. and Traverso, P. (2000). Planning as Model Checking. In Biundo, S. and Fox, M., editors, *Recent Advances in AI Planning*, pages 1–20, Berlin, Heidelberg. Springer Berlin Heidelberg.

[40] Gregory, P., Long, D., Fox, M., and Beck, J. C. (2013). Planning Modulo Theories: Extending the Planning Paradigm. In *Proceedings of the Twenty-Second International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'12, pages 65–73. AAAI Press. event-place: Atibaia, São Paulo, Brazil.

[41] Henzinger, T. A., Nicollin, X., Sifakis, J., and Yovine, S. (1994). Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193 – 244.

[42] Henzinger, T. A., Pei-Hsin Ho, and Wong-Toi, H. (1995). HYTECH: the next generation. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 56–65.

[43] Hoffmann, J. (2003). The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research*, 20:291–341.

[44] Hoffmann, J. and Nebel, B. (2011). The FF Planning System: Fast Plan Generation Through Heuristic Search. *CoRR*, abs/1106.0675.

[45] Hofmann, T. and Lakemeyer, G. (2018). A Logic for Specifying Metric Temporal Constraints for Golog Programs. In *Proceedings of the 11th Cognitive*

Robotics Workshop 2018, co-located with 16th International Conference on Principles of Knowledge Representation and Reasoning, CogRob@KR 2018, Tempe, AZ, USA, October 27th, 2018., pages 36–46.

[46] Hofmann, T., Mataré, V., Schiffer, S., Ferrein, A., and Lakemeyer, G. (2018). Constraint-Based Online Transformation of Abstract Plans into Executable Robot Actions. In *AAAI Spring Symposium 2018 on Integrating Representation, Reasoning, Learning, and Execution for Goal Directed Autonomy*, Stanford, CA, USA.

[47] Holzmann, G. (2003). *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition.

[48] Kautz, H. and Selman, B. (1996). Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, pages 1194–1201. AAAI Press. event-place: Portland, Oregon.

[49] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1998). RoboCup: the Robot World Cup Initiative. *Proceedings of the International Conference on Autonomous Agents*.

[50] Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299.

[51] Kupferschmid, S., Hoffmann, J., Dierks, H., and Behrmann, G. (2006). Adapting an AI Planning Heuristic for Directed Model Checking. In Valmari, A., editor, *Model Checking Software*, pages 35–52, Berlin, Heidelberg. Springer Berlin Heidelberg.

[52] Kupferschmid, S., Wehrle, M., Nebel, B., and Podelski, A. (2008). Faster Than Uppaal? In Gupta, A. and Malik, S., editors, *Computer Aided Verification*, pages 552–555, Berlin, Heidelberg. Springer Berlin Heidelberg.

[53] Lakemeyer, G. and Levesque, H. J. (2004). Situations, Si! Situation Terms, No! In *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning*, KR'04, pages 516–526. AAAI Press. event-place: Whistler, British Columbia, Canada.

[54] Largouët, C., Krichen, O., and Zhao, Y. (2016). Temporal Planning with extended Timed Automata. In Bourbakis, N., Esposito, A., Mali, A., and Alamaniotis, M., editors, *28th International Conference on Tools with Artificial Intelligence (ICTAI 2016)*, SAN JOSE, United States. IEEE.

[55] Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. (1997). GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59 – 83.

[56] Li, Y., Sun, J., Dong, J. S., Liu, Y., and Sun, J. (2012). Planning as Model Checking Tasks. In *2012 35th Annual IEEE Software Engineering Workshop*, pages 177–186.

[57] Matare, V., Schiffer, S., and Ferrein, A. (2018). golog++ : An Integrative System Design. In *Proceedings of the 11th Cognitive Robotics Workshop*.

[58] McCarthy, J. and Laboratory, S. A. I. (1963). *Situations, Actions, and Causal Laws*. Memo (Stanford Artificial Intelligence Project). Stanford University, Artificial Intelligence Project.

[59] Morbé, G., Pigorsch, F., and Scholl, C. (2011). Fully Symbolic Model Checking for Timed Automata. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, pages 616–632, Berlin, Heidelberg. Springer Berlin Heidelberg.

[60] Niebert, P., Tripakis, S., and Yovine, S. (2000). Minimum-Time Reachability for Timed Automata. In *IEEE Mediteranean Control Conference*.

[61] Niemueller, T., Ewert, D., Reuter, S., Ferrein, A., Jeschke, S., and Lakemeyer, G. (2013). RoboCup Logistics League Sponsored by Festo: A Competitive Factory Automation Testbed. In *RoboCup Symposium 2013*.

[62] Niemueller, T., Zug, S., Schneider, S., and Karras, U. (2016). Knowledge-Based Instrumentation and Control for Competitive Industry-Inspired Robotic Domains. *KI - Künstliche Intelligenz*, 30(3):289–299.

[63] Panek, S., Engell, S., and Stursberg, O. (2006a). Scheduling and planning with timed automata. In Marquardt, W. and Pantelides, C., editors, *16th European Symposium on Computer Aided Process Engineering and 9th International Symposium on Process Systems Engineering*, volume 21 of *Computer Aided Chemical Engineering*, pages 1973 – 1978. Elsevier.

[64] Panek, S., Stursberg, O., and Engell, S. (2006b). Efficient synthesis of production schedules by optimization of timed automata. *Control Engineering Practice*, 14:1183–1197.

[65] Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

[66] Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Series in Artificial Intelligence. Prentice Hall, Upper Saddle River, NJ, third edition.

[67] Schupp, S., Abraham, E., Ben Makhlouf, I., and Kowalewski, S. (2017). HyPro: A C++ Library for State Set Representations for Hybrid Systems Reachability Analysis. In *Proc. of the 9th NASA Formal Methods Symposium (NFM'17)*, volume 10227 of *LNCS*, pages 288–294. Springer International Publishing.

[68] Tripakis, S., Yovine, S., and Bouajjani, A. (2005). Checking Timed Büchi Automata Emptiness Efficiently. *Formal Methods in System Design*, 26:267–292.

[69] Wang, F. (2004). Efficient verification of timed automata with BDD-like data structures. *International Journal on Software Tools for Technology Transfer*, 6(1):77–97.