SimuLink to SpaceEx Translator (SL2SX): Final Deliverable.

Stefano Minopoli

UJF Granoble 1 - Lab. VERIMAG, Centre Équation - 2, avenue de Vignate, 38610 GIÉRES
{stefano.minopoli}@imag.fr

1 Introduction

Model-based design is a pervasive approach on developing and implementing complex physical systems. To use this methodology, designers need to build a formal model of the system under design. Then the model will be subject to several extensive analysis in order to establish whether the model effectively shows some desired behaviors (usually described in a formal way by the so-called *specifications*) and, more important, that does not act in a wrong way. The analysis allows to move errors detection to one of the first steps of the design, and this is one of the biggest reasons behind the success of this methodology in the industry. Clearly, to make effective the use of the described approach, there is the need for a formalism to model and analyze the physical system.

The tool *Matlab/Simulink* is the de-facto standard tool used by industry to design and simulate a physical system. Simulink support the model-based design by a block diagram environment for multi-domain simulation, supporting also automatic code generation, and continuous test of embedded systems. Simulink provides a graphical editor, customizable block libraries, and solvers for modeling and simulating dynamic systems. Usually, it is involved in all the developing cycle steps, and this give us an idea of how Simulink and its simulation results play a central role on the real world of the model-based design. For example, in the automotive context, Simulink is first used to mathematically model the plant and the control rules (*Model in the Loop (MiL)* stage). The results of the simulations at MiL level is then compared with the numerical simulation of the (sometimes automatically) obtained control algorithm, written in some language such as C (this is called the *Software in the Loop (SiL)* stage). More, the results of the numerical simulation is then compared with the on-board (or by an emulator) execution of the developed system. It is therefore clear that Simulink, with its numerical simulation, guides the actions of the designers during the different stages of the design and give them enough information about the goodness of the developing and if the design could be accepted or not. Until now we only described the strengths of this industrial approach, but this methodology suffers also for some limits, overall coming from the fact that is simulation-based. Then it is impossible to handle in a correct way every source of non-determinism, like different inputs and noises. More, according to the famous Dijkstra statement, simulation is only able to show the presence of bugs, not even absence.

The central idea of this work comes from the last observation. Our goal is to give to the designers the **mathematically guarantee** that a given design fulfills the specifications (and hence that show the absence of bugs). One of the most common way to achieve this goal is to use the so called *formal verification*, that is the act of proving or disproving the correctness of a system (modeled by the formalism of the *Hybrid Automata* [16]) with respect to a certain formal specification or property. One of the main problems with this approach consists in the fact that usually is very hard to constraint engineers to learn and use new formalisms and tools and to change an established industrial process. Hence, in order to circumvent this problem, we propose a tool that automatically translates a SIMULINK (SL) diagram to an equivalent hybrid automaton that then can be analyzed by a verification tool. In particular, we choose the tool SpaceEx (SX) [12], developed by VERIMAG Laboratory (`http://www-verimag.imag.fr/`), that is the state-of-the-art for verification tools. The performance is not the only thing that motivate our choice to use SpaceEx. By using this tool, we can also achieve other aspects, that are not secondary in terms of the final usability. Indeed, in order to make verification as simple as possible, we want that our translator builds model that is syntactically, semantically and graphically as much similar as possible to the original Simulink Diagram. We could summarize this requirements as follows:

- Variables (name) Preservation: 1-1 correspondence of the variables in the two models
- Graphical Preservation: the graphical shape of the automaton has to be the same of the SL model
- Hierarchical Preservation: the hierarchical structure of the automaton has to be the same of the SL model

SpaceEx allows the hierarchy and the graphical definition of hybrid automata, and then is the best choice for both the aspects, performance and usability.

Once decided that the target formalism is Hybrid Automata, it is fundamental to establish its relationship with the formalism of Simulink in terms of expressive power. In other words, we have to answer to the question about how much is possible to express a Simulink diagram by using the formalism of hybrid automata. The answer resides in fact that Simulink (as weel as other tool like Modelica [20] or Ptolemy [8]) is defined with *Must Semantics*. As a consequence, the evolution of the system is deterministic (the discrete transitions are taken as soon as they are enabled. Also referred as urgent or ASAP transitions). This can pose a problem when one tries to build a corresponding hybrid model, because here transitions do not force the system to change state when they are enabled. For example, consider the case when the derivatives of a system variable is zero when the guard is enabled: an hybrid automaton may remain forever at that state.

To fix this gap between the two formalisms, we allow to couple location with *Urgent-Condition*: in this way asap transitions became very easy to model by hybrid automata. Clearly, we introduce a novel sound a complete algorithm to

compute the reachable set of valuations inside a location with a urgent condition. The theoretical results, and the implementation on the SpaceEx platform, were presented in the *12-th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2014)* [21].

The effective implementation of the urgency conditions in LHA, allows us to direct translate the Simulink dicrete blocks (i.e. Switch block, DeadZone block and ot.) to SpaceEx basic component. Hence, we have a way to perform formal verification on Simulink diagrams via SL2SX translation tool.

The rest of this work is organized as follows. Section 2.1 introduces and motivate the formalism of Linear Hybrid Automata enhanced by urgency conditions. In Section 3, we describe the syntax and the semantics of Simulink and the eXtensible Markup Language (XML) representation, while Section 4 describes in details the tool SpaceEx and the XML representation of a SpaceEx model. Section 5 reports how the translator SL2SX is designed and implemented. Finally, Section 6 show a concrete use case, starting from a Simulink diagram to the reachability analysis on the obtained SpaceEx model.

2 Linear Hybrid Automata (LHAs) and Urgency Conditions

An Hybrid Automata (HAs) models a system essentially by (i) a finite set of locations (and theirs transitions) to model the discrete evolution, and (ii) a finite set of continuous variables, governed by differential equations, to model the continuous evolution.

In particular, in this work we are focused on the subclass of *Linear Hybrid Automata (LHA)* [16], for two primary reasons. First, despite their syntactical simplicity, they admit a rich variety of behaviors. In LHA, the evolution of the variables over time is governed by differential inclusions, called *flows*, which can be simple intervals such as $\dot{x} \in [1, 2]$, or more complex linear constraints over the derivatives such as the conservation law $\dot{x} + \dot{y} = 0$. Changes of the discrete state admit arbitrary linear updates of the variables. For example, LHA can model discrete-time affine systems, a widely used class of control systems, by using discrete updates of the form $x^+ = Ax + b$. LHA can even generate chaotic behavior that can be observed in real-life production systems [24]. The chaos can be due to switching flows [9] or due to updates of the discrete state, with which one can model piecewise affine maps such as the tent map [10]. The last one motivation in because LHA belong to the very few classes of hybrid systems for which set-based successor computations can be carried out exactly [1]. This makes them prime candidates for formal verification. LHA can serve as abstractions of systems that require not only timed behavior but quantitative information, e.g., to capture accumulation effects. The LHA abstraction can then be verified using model checkers such as HyTech [15] or PHAVer [13]. If the abstraction is conservative, verifying it implies that the real system satisfies the specification; if the abstraction is an approximation that is not entirely conservative, its verification helps to find bugs and identify pertinent test cases.

In model-based design, the basis for building LHA abstractions is often an existing model, given in formats like Matlab-Simulink/Stateflow [19, 18] or Modelica [20], which are the de-facto standard in many industries. Like the academic formalism Ptolemy [8], the semantics of these models are deterministic, also referred as *Must Semantics*. In particular, a discrete transition is taken as soon as it is enable (urgent or ASAP transitions). This can pose a problem when one tries to build a corresponding LHA model, since LHA transitions do not force the system to change state when they are enabled. For example, consider the case when the derivatives of a system variable is zero when the guard is enabled: in LHA the system may remain forever at that state (this is called the *May Semantics* of the HA). The only way to circumvent this problem is to add a clock to the controller model and periodically test (with a self-loop transition) whether the constraint is satisfied or not. This is a formally correct and conservative way to model such a system, and it even corresponds quite closely to actual behavior of process controllers, which periodically sample the sensors and set actuators. But it can tremendously increase the computational complexity of the verification task: The clock ticks introduce discrete state changes at a rate much higher than the time constants of the system, multiplying the number of sets of states that need to be computed. A computationally more efficient abstraction can be obtained if one adds *urgency* conditions to the LHA formalism. Declaring certain states of the controller as urgent prevents time from elapsing, and one can now construct an LHA abstraction (or approximation) of deterministic transitions.

Existing algorithms for set-based successor computations of LHA require urgency conditions to either be independent of the continuous variables [15] or consist of a single constraint [13], which can be quite restrictive in practice. In this paper, we propose an algorithm to compute successor states for arbitrary, non-convex, closed urgency conditions. To be able to do so, we also propose an algorithm for computing successor states for general non-convex invariants, for which so far no algorithm is available. Related work is discussed in more detail for non-convex invariants in Sect. 2.2 and for urgency in Sect. 2.4.

The proposed algorithms are implemented in the open-source tool PHAVer on the SpaceEx tool platform [12]. The tool as well as all examples from this paper are available for download at `spaceex.imag.fr`. Detailed proofs are available in a technical report [22].

By introducing urgency in Linear Hybrid Automata, we fix the conceptual mismatch between the semantics of Simulink and Hybrid Automata: this pave the way to design an automatic translator from a model described by Simulink to an equivalent hybrid automaton. The possibility to design such a tool is very important because, considering how Simulink is pervasive used in the industry, it is not plausible to constraint designers to directly use one of the several academic tools developed for the verification of hybrid automata. Hence, the availability of an automatic translator may make very easy and effective the usage of the formal verification, without the need to acquire new knowledge.

2.1 Linear Hybrid Automata with Non-Convex Invariants

In this section, we give the syntax and the semantics description of a particular case of *Linear Hybrid Automata (LHA)*, where it is possible to define, for each location, a non-convex invariant.

Definition and Semantics We first need to define some notation. A *convex polyhedron* is a subset of $\mathbb{R}^n$ that is the intersection of a finite number of strict and non-strict affine half-spaces. A *polyhedron* is a subset of $\mathbb{R}^n$ that is the union of a finite number of convex polyhedra. For a sake of clearity, given a convex polyhedron P, we will write $\widehat{P}$ to direct explicit that P is convex. For a general (i.e., not necessarily convex) polyhedron $G \subseteq \mathbb{R}^n$, we denote by $cl(G)$ its topological closure. Given an ordered set $X = \{x_1, \ldots, x_n\}$ of variables, a *valuation* is a function $v : X \to \mathbb{R}$. Let $Val(X)$ denote the set of valuations over X. There is an obvious bijection between $Val(X)$ and $\mathbb{R}^n$, allowing us to extend the notion of (convex) polyhedron to sets of valuations. We denote by $CPoly(X)$ (resp., $Poly(X)$) the set of convex polyhedra (resp., polyhedra) on X. We use $\dot{X}$ to denote the set $\{\dot{x}_1, \ldots, \dot{x}_n\}$ of dotted variables, used to represent the first derivatives, and X' to denote the set $\{x'_1, \ldots, x'_n\}$ of primed variables, used to represent the new values of variables after a discrete transition. Arithmetic operations on valuations are defined in the straightforward way. An *activity* over X is a function $f : \mathbb{R}^{\geq 0} \to Val(X)$ that is continuous on its domain and differentiable except for a finite set of points. Let $Acts(X)$ denote the set of activities over X. The *derivative* $\dot{f}$ of an activity f is defined in the standard way and it is a partial function $\dot{f} : \mathbb{R}^{\geq 0} \to Val(\dot{X})$.

A *Linear Hybrid Automaton* $H = (Loc, X, Lab, Edg, Flow, Inv, Init)$ consists of the following:

- a finite set Loc of *locations*,
- a finite set $X = \{x_1, \ldots, x_n\}$ of real-valued *variables*, A *state* is a pair $\langle l, v \rangle$ of a location l and a valuation $v \in Val(X)$;
- a finite set of labels Lab,
- a finite set Edg of *discrete transitions* that describes instantaneous changes of locations, in the course of which variables may change their value. Each transition $(l, \alpha, \eta, l') \in Edg_c$ consists of a *source location* l, a *target location* l', a label $\alpha \in Lab$, and a *jump relation* $\eta \in Poly(X \cup X')$, that specifies how the variables may change their value during the transition. The *guard* is the projection of η on X and describes the valuations for which the transition is enabled;
- a mapping $Flow : Loc \to CPoly(\dot{X})$ attributes to each location a set of valuations over the first derivatives of the variables, which determines how variables can change over time;
- a mapping $Inv : Loc \to Poly(X)$, called the *invariant*;
- a mapping $Init : Loc \to Poly(X)$, contained in the invariant, defining the *initial states* of the automaton.

The set of states of H is $S = Loc \times Val(X)$. Moreover, we use the shorthand notations $InvS = \bigcup_{l \in Loc} \{l\} \times Inv(l)$ and $InitS = \bigcup_{l \in Loc} \{l\} \times Init(l)$. Given a set of states A and a location ℓ, we denote by $A \downarrow_\ell$ the projection of A on ℓ, i.e. $A \downarrow_\ell = \{v \in Val(X) \mid \langle \ell, v \rangle \in A\}$.

Semantics The behavior of a LHA is based on two types of steps: *discrete* steps correspond to the *Edg* component, and produce an instantaneous change in both the location and the variable valuation; *timed* steps describe the change of the variables over time in accordance with the *Flow* component.

Given a state $s = \langle l, v \rangle$, we set $loc(s) = l$ and $val(s) = v$. An activity $f \in Acts(X)$ is called *admissible from* s if *(i)* $f(0) = v$ and *(ii)* for all $\delta \geq 0$, if $\dot{f}(\delta)$ is defined then $\dot{f}(\delta) \in Flow(l)$. An activity is *linear* if there exists a constant slope $c \in Flow(l)$ such that, for all $\delta \geq 0$, $\dot{f}(\delta) = c$. We denote by $Adm(s)$ the set of activities that are admissible from s.

Runs Given two states s, s', and a transition $e \in Edg$, there is a *discrete step* $s \xrightarrow{e} s'$ with *source* s and *target* s' iff *(i)* $s, s' \in InvS$, *(ii)* $e = (loc(s), \alpha, \eta, loc(s'))$, and *(iii)* $(val(s), val(s')[X'/X]) \in \eta$, where $val(s')[X'/X]$ is the valuation in $Val(X')$ obtained from s' by renaming each variable in X with the corresponding primed variable in X'. Whenever condition *(iii)* holds, we say that e is *enabled* in s. There is a *timed step* $s \xrightarrow{\delta, f} s'$ with *duration* $\delta \in \mathbb{R}^{\geq 0}$ and activity $f \in Adm(s)$ iff *(i)* $s \in InvS$, *(ii)* for all $0 < \delta' \leq \delta$, $(\langle l, f(\delta') \rangle) \in InvS$, and *(iii)* $s' = \langle loc(s), f(\delta) \rangle$. For technical convenience, we admit timed steps of duration zero. A special timed step is denoted by $s \xrightarrow{\infty, f}$ and represents the case when the system follows the activity f forever. This is allowed only if for all $\delta \geq 0$, $(\langle l, f(\delta) \rangle) \in InvS$. A *run* is a sequence

$$r = s_0 \xrightarrow{\delta_0, f_0} s'_0 \xrightarrow{e_0} s_1 \xrightarrow{\delta_1, f_1} s'_1 \xrightarrow{e_1} s_2 \cdots s_n \cdots \tag{1}$$

of alternating timed and discrete steps, such that either the sequence is infinite, or it ends with a timed step of the type $s_n \xrightarrow{\infty, f}$.

If the run r is finite, we define $len(r) = n$ to be the length of the run, otherwise we set $len(r) = \infty$. Given a hybrid automaton H, the set $Runs(H)$ contains all the possible runs induced by the automaton H.

Reachability Given a state $s \in S$ and a hybrid automaton H with initial set of states $Init$, s is said to be *reachable* in H if there exists a finite run $r = s_0 \xrightarrow{\delta_0, f_0} s'_0 \xrightarrow{e_0} s_1 \xrightarrow{\delta_1, f_1} s'_1 \xrightarrow{e_1} s_2 \cdots s_n$, such that $s_0 \in Init$ and $s_n = s$. In a natural way, we can extend the concept of reachability to the valuations: considering the run r explicited before, if $s_0 = (\langle l, u \rangle)$ and $s_n = (\langle l', v \rangle)$, we can say that the valuation v is reachable from the valuation u. The *reachability problem* for the automaton H consists in the computation of the set of states

$$Reach(H) = \big\{ s \in S \,\big|\, \exists r \in Runs(H) : len(r) \neq \infty, r = s_0 \xrightarrow{\delta_0, f_0} \cdots s_n, \text{ where } s_0 \in Init \text{ and } s_n = s \big\}.$$

Classically, the algorithm that computes the set $Reach(H)$ is a fixed-point procedure, over all the locations $l \in Loc$, based on the *continuous post operator* and on the *discrete post operator*: given a set of states $S' \subseteq S$, the first one operator is used to compute the set of states reachable from S' by following an admissible trajectory, while the second one operator is used to compute the set of states reachable from S' via discrete transitions. Notice that the computation of the discrete post operator is not affected by the nature of the invariants, so we focus on the continuous post operator. The formal definitions are as follows:

Definition 1 (Post operators). *Given an hybrid automaton H, a location $\ell \in Loc$, a set of valuations $P, I \subseteq Inv(\ell)$, the* continuous post operator *$Post_\ell(P, I)$ contains the set of all valuations $v \in Val(X)$ reachable from some $u \in P$ without leaving I:*

$$Post_\ell(P,I) = \big\{ v \in val(X) \big| \exists u \in P, f \in Adm(\langle l, u\rangle) \text{ and } \delta \geq 0 : \\ \forall 0 < \delta' \leq \delta, f(\delta') \in I \quad \text{and } f(\delta) = v \big\}. \quad (2)$$

The discrete post operator *$Post_\varepsilon(P)$ contains the set of all valuations $v \in Val(X)$ reachable from some $u \in P$ by taking the discrete transition $\varepsilon = (\ell, \eta, \ell')$:*

$$Post_\varepsilon(P) \;=\; \big\{ v \;\in\; val(X) \big| \exists u \;\in\; P, (u, v[X'/X]) \;\in\; \eta \text{ and } v \;\in\; Inv(\ell') \big\}.$$

From these operators on valuations we obtain the continuous and discrete post operators for a set of states S by iterating over all locations and transitions:

$$Post_c(S) = \bigcup_{\ell \in Loc} \{\ell\} \times Post_\ell(S\lfloor_\ell, Inv(\ell)), \quad Post_d(S) = \bigcup_{(\ell,\alpha,\eta,\ell') \in Edg} \{\ell'\} \times Post_\varepsilon(S\lfloor_\ell).$$

Note that definition (2) is valid regardless whether I is convex or not. It differs slightly from the classic definition in that we do not require that $P \subseteq I$. This trick is used in the next section to apply the operator iteratively to convex partitions of a non-convex invariant. In this case, I is a convex subset of the invariant but P is not necessarily a subset of I. For the sake of clarity, we will denote by $Post_\ell(P, I)$ the continuous post operator when I is convex and by $ncPost_\ell(P, I)$ when I is non-convex.

The following simple algorithm is used by tools such as HyTech and PHAVer to compute the reachable states. Starting from the initial states, it computes the continuous and discrete post in alternation until a fixed point is reached. Note that this is a semi-algorithm, since it may not terminate. The algorithm computes the sequence $S_0 = Post_c(InitS)$, and $S_{k+1} = S_k \cup Post_c(Post_d(S_k))$. It terminates if $S_{k+1} = S_k$, with the result that $Reach(H) = S_k$.

2.2 Computing the Continuous Post Operator with Nonconvex Invariants

In this section we first recall how the continuous post operator is computed, when the invariant is a convex polyhedron, and why it is not possible to use the

same way when the invariant is non-convex. Then we give a sound and complete procedure that, given a non-convex invariant I and an initial set of valuation $P \subseteq I$, computes the continuous post operator $ncPost_\ell(P, I)$. Given a linear hybrid automaton H, it is well known that the continuous post operator, on a location $l \in Loc$, a convex invariant $I = Inv(\ell)$, a flow $F = Flow(\ell)$ and a set of initial valuations $P \subseteq Inv(\ell)$, is given by:

$$Post_\ell(P, I) = (P \nearrow_F) \cap I, \tag{3}$$

where $P \nearrow_F$ are valuations on straight line trajectories starting in P with constant derivative $\dot{x} = c$ for any $c \in F$:

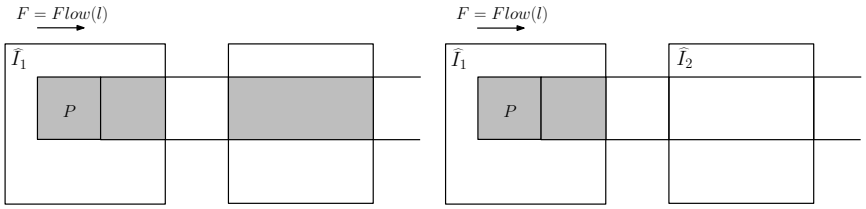
$$P \nearrow_F = \{x' \mid x \in P, c \in F, t \in \mathbb{R}^{\geq 0}, x' = x + ct\}. \tag{4}$$

The operator (4) is straightforward to compute for polyhedral sets, and is available in computational geometry libraries such as the Parma Polyhedra Library (PPL) [2]. The result of (4) is a convex polyhedron if P, F are convex polyhedra and F is closed and bounded; otherwise, it is the union of P with a convex polyhedron [7].

The natural question now is: what happens if (3) is applied when I is a non-convex polyhedron? Example 1 illustrates the answer with a simple case.

Example 1. Given a LHA with non-convex polyhedra invariants $H = (Loc, X, Edg, Flow, Inv, Init)$, and fix $l \in Loc$, $I = \widehat{I}_1 \cup \widehat{I}_2 = Inv(\ell)$, $P \subseteq I$, and $F = Flow(\ell)$, Figure 1 shows the comparison between the correct result of $ncPost_\ell(P, I)$ and the result obtained by forcing the usage of the classical post operator also with non-convex invariant (that is by computing $P \nearrow_F \cap I$). The gray area in Figure 1(a) contains all the valuations coming from (3). Notice that, in the resulting set there are also valuations belonging to $\widehat{I}_2$. But, according to the definition of continuous post, this is not correct because all the valuations reached during the evolution of the time have to remain in the invariant: in this case, for all the admissible activities in P, the only way to reach $\widehat{I}_2$, is to cross the area between $\widehat{I}_1$ and $\widehat{I}_2$, that is clearly not in the invariant and then the valuations in $Post_\ell(P, I) \cap \widehat{I}_2$ do not belong to $ncPost_\ell(P, I)$. The gray area of Figure 1(b) contains, instead, all the valuations that properly belong to the $ncPost_\ell(P, I)$: starting from any valuations in P, the evolution can proceed only until the right border of $\widehat{I}_1$ is reached, and there is no way to reach $\widehat{I}_2$ without crossing $\overline{I}$.

Example 1 can also give an intuition about how to tackle the issues of the classical approach: in order to compute $ncPost_\ell(P, I)$, it is necessary to consider not the global I, but the single convex polyhedra in $[\![I]\!]$, in a kind of fix-point procedure. Considering again Figure 1, the first step is to compute $P_1 = P \nearrow_F \cap \widehat{I}_1$. Once obtained the result, we need to check if there exists a valuation $u \in P_1$ such that, by following some admissible activity $f \in Adm(\langle \ell, u \rangle)$, it is possible to reach a valuation $v \in \widehat{I}_2$, while the system always remains in the invariant (i.e. while avoiding $\overline{I}$). In the case depicted in Figure 1, does not exist such a valuation



(a) $(P\nearrow_F) \cap I$. (b) The correct result.

Fig. 1. Comparison between $Post_\ell(P, I)$ and $ncPost_\ell(P, I)$, with non-convex invariant $I = Inv(\ell) = \widehat{I}_1 \cup \widehat{I}_2$.

(indeed, in order to reach $\widehat{I}_2$ via an admissible activity, it is always necessary to cross $\overline{I}$).

Before giving the fixed point characterization of $ncPost_\ell$, we need to introduce some extra notation (some of them similar to operators defined in [7]). The topological closure of a set G is denoted by $cl(G)$. Given polyhedra A and B, their *boundary* is

$$bndry(A, B) = (cl(A) \cap B) \cup (A \cap cl(B)). \tag{5}$$

Clearly, $bndry(A, B)$ is nonempty only if A and B are adjacent to one another or they overlap; otherwise, it is empty.

Definition 2 (Potential entry). *Given a location ℓ and convex polyhedra A and B, the potential entry region from A to B denotes the set of points on the boundary between A and B that may reach B by following some linear activity in location ℓ, while always remaining in $A \cup B$:*

$$pentry_\ell(A, B) = \big\{p \in bndry(A, B) \mid \exists q \in A, \delta \geq 0 \text{ and } c \in Flow(\ell) : \\ p = q + \delta \cdot c \text{ and for all } 0 \leq \delta' < \delta,\ q + \delta' \cdot c \in A\big\}. \tag{6}$$

We call the above set the "potential" entry because it may happen that, even though $pentry_\ell(A, B)$ is not empty, the system is not able to reach valuations in B starting from a valuation in A (see Example 2, Fig. 2(c), Fig. 2(f)). The following Lemma gives us a way to effectively compute the potential entry region.

Lemma 1. *Given a location ℓ and convex polyhedra A and B, let $F = Flow(\ell)$, the potential entry region from A to B can be computed by:*

$$pentry_\ell(A, B) = bndry(A, B) \cap A\nearrow_F .$$

Proof ($\subseteq$). Let $p \in pentry_\ell(A, B)$, by definition of potential entry we have that $p \in bndry(A, B)$ and there exists a valuation $q \in A$, $c \in F$ and a time $\delta \geq 0$, such that $p = q + \delta \cdot c$ and for all $0 \leq \delta' < \delta$, it holds that $q + \delta' \cdot c \in A$. The last one, recalling that $q \in A$ and the time elapse-definition, allows us to write that for all $0 \leq \delta' < \delta$, it holds that $q + \delta' \cdot c \in A\nearrow_F$. Again, by definition of time elapse,

if for all $0 \leq \delta' < \delta$, then $q + \delta' \cdot c \in A\nearrow_F$ and also $p = q + \delta \cdot c \in A\nearrow_F$. Hence, $p \in bndry(A, B)$ and $p \in A\nearrow_F$ trivially implies that $p \in bndry(A, B) \cap A\nearrow_B$.

[$\supseteq$] Let $p \in bndry(A, B) \cap A\nearrow_F$, by definition of time-elapse we have that there exists a valuation $q \in A$, a point $c \in F$ and a time $\delta \geq 0$ such that $p = q + \delta \cdot c$ or, equivalently there exists an activity $f \in Adm(\langle l, u \rangle)$ such that $p = f(\delta) \in bndry(A, B)$. Due to the convexity of A and the definition of boundary, we have that for all $0 \leq \delta' < \delta$, $f(\delta') \in A$. The last one, coupled with $f(\delta) \in bndry(A, B)$, allows us to conclude $p \in pentry_\ell(A, B)$. □

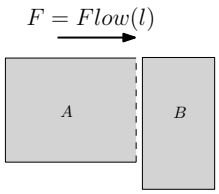
From Lemma 1 follows the following Corollary:

Corollary 1. *If $A \subseteq B$, then $A \subseteq pentry_\ell(A, B) \subseteq cl(A)$.*

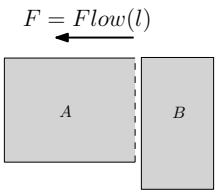
Example 2. Figure 2 shows two convex polyhedra A and B whose boundary is non empty (A and B are adjacent), where flow is represented by an arrow. Considering Figure 2(a), it is easy to check that the flow allows to reach valuations on the boundary between A and B starting from a valuation belongs to A. Figure 2(d) shows the computation of $pentry_\ell(A, B)$, obtained by performing the intersection between $bndry(A, B)$ and $A\nearrow_F$. Considering instead the case depicted in Figure 2(b) (same as the previous one except for the flow), there is no way to reach valuations belong to $bndry(A, B)$ starting from a valuation $u \in A$: according to Lemma 1, Figure 2(e) shows that the intersection between the boundary and the post-flow is empty. Figure 2(c) shows a case where the polyhedron B is not closed and then by following the flow, the system can never reach B, even if the starting valuation is on the top border of A. Notice that (see Figure 2(f)), even if B is not reachable from A, we have that $pentry_\ell(A, B) \neq \emptyset$: this clearifies why we denote this set as "potential".

Now we are ready to give a way to correctly compute the continuous post operator when the invariants could be non-convex. Given a LHA H and let $l \in Loc$, $I = Inv(\ell)$, $F = Flow(\ell)$ and $P \subseteq I$, as Example 1 suggests, the idea is to build incrementally the sets of reachable valuations by considering each time a single convex component $\widehat{I}' \in [\![I]\!]$ instead of considering the entire invariant I. The procedure starts by finding, for all $\widehat{I}' \in [\![I]\!]$ and $\widehat{P}' \in [\![P]\!]$, the potential entry from $\widehat{P}'$ to $\widehat{I}'$. Once obtained the set $pentry_\ell(\widehat{P}', \widehat{I}')$, the procedure computes the classical continuous post operator on $pentry_\ell(\widehat{P}', \widehat{I}')$ and $\widehat{I}'$. The procedure is applied recursively by building the sequence $W_0 \subseteq W_1 \subseteq \ldots W_{i-1} = W_i$ of the sets of the reachable valuations, with $W_0 = P$, and ends when no new valuation can be added to a set. When this happens, we have that $ncPost_\ell(P, I) = W_i$.

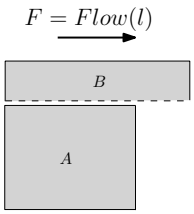
Before formalizing we informally explain, by Figure 3, why the procedure needs to compute the potential entry sets $pentry_\ell(\widehat{W}', \widehat{I}')$, instead of simply performing the intersection between $\widehat{W}'$ and $\widehat{I}'$. Consider the step that build the set $\widehat{W}_3$, as depicted in Figure 3(b), and suppose that the procedure, instead of using $Post_\ell(pentry_\ell(\widehat{W}_2, \widehat{I}_3), \widehat{I}_3)$ would use $Post_\ell(\widehat{W}_2 \cap \widehat{I}_3, \widehat{I}_3)$. Due to the fact that $\widehat{I}_2$ is right open, also $\widehat{W}_2$ is right open and then the initial set $\widehat{W}_2 \cap \widehat{I}_3$ would be empty as well as $Post_\ell(\widehat{W}_2 \cap \widehat{I}_3, \widehat{I}_3)$.



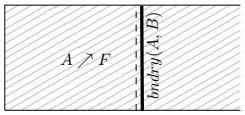
(a) Case 1: the flow allows to reach B from $bndry(A, B)$.



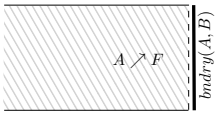
(b) Case 2: the flow does not allow to reach B from $bndry(A, B)$.



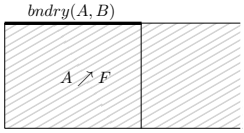
(c) Case 3: the flow does not allow to reach B from $bndry(A, B)$.



(d) $bndry(A, B) \cap A \nearrow_F \neq \emptyset$: potential entry from A to B.



(e) $bndry(A, B) \cap A \nearrow_F = \emptyset$: entry from A to B impossible.



(f) $bndry(A, B) \cap A \nearrow_F \neq \emptyset$, but the system can not reach B from A.

Fig. 2. The computation of potential entry from A to B involves computing the boundary of A and B (as shown in Figs. 2(a),2(b),2(c)), and identifying states reachable on that boundary (see, respectively, Figs. 2(d),2(e),2(f))

Notice that, even if the initial set of valuations $pentry_\ell(\widehat{W}_2, \widehat{I}_3)$ is not in the invariant convex component $\widehat{I}_3$, the set $Post_\ell(pentry_\ell(\widehat{W}', \widehat{I}')$ is non-empty. We have that $pentry_\ell(\widehat{W}_2, \widehat{I}_3) \subseteq I$ that is the starting valuations have to belong to the global non-convex invariant I, not necessary to the convex component $\widehat{I}_3 \in [\![I]\!]$.

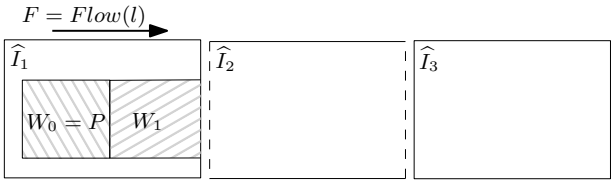
The formal relationship between the fixed-point procedure described above and the computation of the continuous post operator, when the invariant is non-convex, is given by the following theorem:

Theorem 1. *Given a location $\ell \in Loc$ and sets $P \subseteq Inv(\ell)$, $I = Inv(\ell)$, $ncPost_\ell(P, I)$ is the smallest fixed point of the sequence $W_0 = P$,*

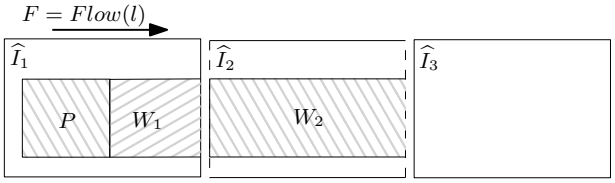
$$W_k = \bigcup_{\widehat{W}' \in [\![W_{k-1}]\!]} \bigcup_{\widehat{I}' \in [\![I]\!]} Post_\ell(pentry_\ell(\widehat{W}', \widehat{I}'), \widehat{I}').$$

Moreover, the above sequence reaches the fixed point in at most $n = \left|[\![I]\!]\right|$ steps, that is $W_{n+1} = W_n$.

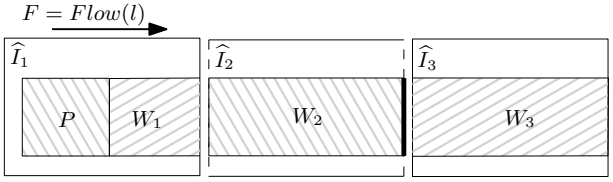
We split the proof of Theorem 1 into several lemmas. Given a polyhedron I and two valuations u and v, assuming that v is reachable from u via an admissible activity that always remains in I (i.e. always avoids $\overline{I}$), we denote by $d(u, v, I)$



(a) First step: computation of W_1 starting from the initial set P.



(b) Second step: computation of W_2 starting from $pentry_\ell(W_1, \widehat{I}_2)$.



(c) Last step: computation of W_3 starting from $pentry_\ell(W_2, \widehat{I}_3)$.

Fig. 3. The central role of $pentry_\ell(W_2, \widehat{I}_3)$ in the computation of $ncPost_\ell(P, I)$.

the minimum number of convex polyhedra in $[\![I]\!]$ in which the system must remain in order to reach v from u via any admissible activity f. Formally:

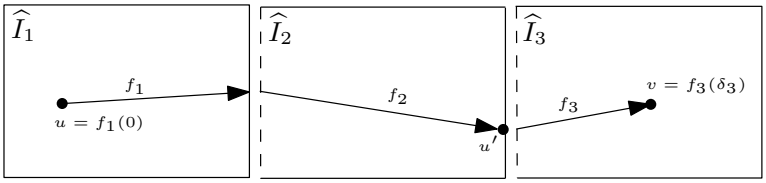
$$d(u, v, I) = \min\{n > 0 \mid \exists f \in Adm(\langle \ell, u \rangle), \delta \geq 0, \widehat{I}_1, \dots \widehat{I}_n \in [\![I]\!] : \\ f(\delta) = v \text{ and } \forall 0 \leq \delta' \leq \delta \, \exists j \in \{1, \dots, n\} : f(\delta') \in \widehat{I}_j\}.$$

When there is no activity that can reach v from u avoiding $\overline{I}$, we write $d(u, v, I) = \infty$. Hence either $d(u, v, I) \leq \left|[\![I]\!]\right|$ or $d(u, v, I) = \infty$. Now, we define a slightly different version of $ncPost_\ell$ that take into account only valuations v such that the system, in order to reach v, always remains in a fixed number of convex polyhedra in the invariant. Formally, given a location l and fixed $I = Inv(\ell)$, $P \subseteq I$ and $i \leq |[\![I]\!]|$,

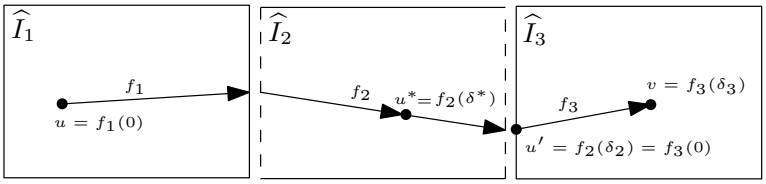
$$ncPost_\ell(P, I, i) = \big\{v \in ncPost_\ell(P, I) | \exists u \in P : d(u, v, I) \leq i\big\}.$$

Note that for all $i \leq j$, $ncPost_\ell(P, I, i) \subseteq ncPost_\ell(P, I, j)$.

A fundamental property of LHA is that if there is an activity that goes from u to v inside the invariant, there is also a sequence of linear activities that does the same. Moreover, each linear activity is contained within one convex polyhedron of $[\![I]\!]$ and hence the connecting points between any two consecutive linear activities lie on the boundary between two polyhedra in $[\![I]\!]$. The following formalization is a reformulation of Lemma 2.2 in [26] given as Lemma 5 in [7]:



(a) From u to u' the system never touches $\widehat{I}_3$.



(b) From u to u' the system touches also $\widehat{I}_3$.

Fig. 4. Cases (a) and (b) of the proof of Lemma 3.

Lemma 2. [7] *Let u and v be valuations, and I a polyhedron. If $d(u, v, I) = i < \infty$, then there is a sequence of linear activities $f_1, \ldots, f_i$, delays $\delta_0, \ldots, \delta_i$, and convex polyhedra $\widehat{I}_1, \ldots, \widehat{I}_i \in [\![I]\!]$ such that* (i) *$f_1 \in Adm(\langle l, u\rangle)$,* (ii) *$f_{i-1}(\delta_{i-1}) = v$,* (iii) *for all $j < i$ it holds $f_j(\delta_j) \in bndry(\widehat{I}_j, \widehat{I}_{j+1})$ and $f_{j+1} \in Adm(\langle l, f_j(\delta_j)\rangle)$, and* (iv) *for all $j \leq i$ and $0 < \delta' < \delta_j$ it holds $f_j(\delta') \in \widehat{I}_j$.*

Now we are ready to give two lemma that prove Theorem 1.

Lemma 3. *For all locations ℓ, polyhedra P and I, and $i \geq 1$, it holds*

$$ncPost_\ell(P, I, i) \subseteq W_i.$$

Proof. Let v be a valuation belonging to $ncPost_\ell(P, I, i)$, by definition we have that there exists a valuation $u \in P$, an activity $f \in Adm(\langle \ell, u\rangle)$ and a time $\delta \geq 0$ such that $f(\delta) = v$, $d(u, v, I) = i$ and, for all $0 < \delta' \leq \delta$, $f(\delta') \in I$.

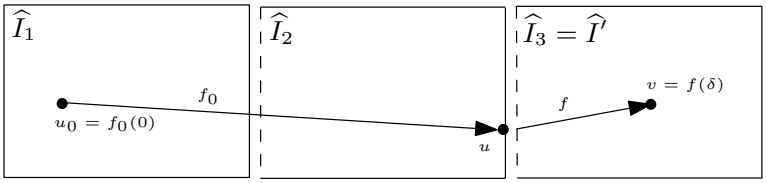
Now, we proceed by induction on $i \geq 1$. For the base case, if $i = 1$ then $d(u, v, I) = 1$. Recalling the definition, $d(u, v, I) = 1$ means that in order to reach valuation v from u the system remains always in just one convex component of $[\![I]\!]$. Hence, there exists a convex polyhedron $\widehat{I}' \in [\![I]\!]$ such that, for all $0 \leq \delta' \leq \delta$, it holds that $f(\delta') \in \widehat{I}'$. As a consequence, $f(0) = u \in P \cap \widehat{I}'$ and, by Corollary 1, we obtain $u \in pentry_\ell(P, \widehat{I}')$. From the valuation u that belongs to the potential entry from P to $\widehat{I}'$, it is possible to reach the valuation v via the activity f by always remaining in $\widehat{I}'$. But this is the definition of post operator, and then we can write $v \in Post_\ell(pentry_\ell(P, \widehat{I}')) \subseteq W_1$, ending the base case. For the inductive step, we consider $i \geq 2$. If the valuation v also belongs to $ncPost_\ell(P, I, i-1)$ then the thesis trivially holds by inductive hypothesis. Otherwise, by Lemma 2, there exists a sequence of activities f_1, ..., f_i, of convex component I_1, ..., $I_i \in [\![I]\!]$ and times δ_1, ..., δ_i, such that from valuation $u = f_1(0)$ it is possible

to reach the valuation $u' = f_{i-1}(\delta_{i-1})$, and $u' = f_{i-1}(\delta_{i-1}) \in bndry(\widehat{I}_{i-1}, \widehat{I}_i)$. Moreover, Lemma 2 also says that by following activities $f_1, \ldots, f_{i-1}$, the system always remains in the convex components $I_1, \ldots, I_{i-1}$ and in $bndry(\widehat{I}_{i-1}, \widehat{I}_i)$. By definition of boundary, we have either (a) $u' \in \widehat{I}_{i-1} \cap cl(\widehat{I}_i)$ (and then the system always remains in $I_1, \ldots, I_{i-1}$) or (b) $u' \in \widehat{I}_i \cap cl(\widehat{I}_{i-1})$ (and then the system always remains in $I_1, \ldots, I_i$. Considering case (a), since the system always remains in $I_1, \ldots, I_{i-1}$, it is clear that $d(u, u', I) = i - 1$ and then, by inductive hypothesis, $u' \in W_{i-1}$. Hence, $u' \in W_{i-1} \cap cl(\widehat{I}_i)$, that means that there exists a convex polyhedron $\widehat{W}' \in [\![W_{i-1}]\!]$ such that $u' \in \widehat{W}' \cap cl(\widehat{I}_i) \subseteq \widehat{W}'\nearrow_F$ and clearly $u' \in bndry(\widehat{W}', \widehat{I}_i)$. Considering the last two conditions and Lemma 1, we have that $u' \in pentry_\ell(\widehat{W}', \widehat{I}_i)$. Now, from the valuation $u' \in pentry_\ell(\widehat{W}', \widehat{I}_i)$, it is possible to reach the valuation v via the activity f_i without leaving the invariant $\widehat{I}_i$, allowing us to write $v \in Post_\ell(pentry_\ell(\widehat{W}', \widehat{I}_i)) \subseteq W_i$, and this ends the first case (see Figure 4(a) to better understand the reasoning). Case (b), otherwise, is when $u' \in \widehat{I}_i \cap cl(\widehat{I}_{i-1})$ and then in order to reach the valuation u' from u the system always remains in $I_1, \ldots, I_i$. By Lemma 2, for all $0 < \delta^* < \delta_{i-1}$, the valuation $u^* = f_{i-1}(\delta^*)$ belong to $\widehat{I}_{i-1}$ and is reachable from u while the system always remains in $\widehat{I}_1, \ldots, \widehat{I}_{i-1}$. Hence, by definition, $d(u, u^*, I) = i - 1$ and then $u^* \in ncPost_\ell(P, I, i-1)$. By inductive hypothesis, we have also that $u^* \in W_{i-1}$, that is there exists $\widehat{W}' \in W_{i-1}$ such that $u^* \in \widehat{W}'$. Moreover, u' can be reached from any of the u^* via the activity f_{i-1}. But this means that $u' \in \widehat{W}'\nearrow_F$ and, recalling that $u' \in \widehat{I}_i$, this means that $u' \in pentry_\ell(\widehat{W}', \widehat{I}_i)$ (Lemma 1). Now, by using again Lemma 2, there exists the activity f_i and the time δ_i such that (i) $f_i(0) = u'$, (ii) $f_i(\delta_i) = v$, and (iii) for all $0 < \delta' < \delta$, $f_i(\delta') \in \widehat{I}_i$. By combining conditions i, ii and iii, we have that $v \in Post_\ell(pentry_\ell(\widehat{W}', \widehat{I}_i), \widehat{I}_i) \subseteq W_i$. This ends the second case (see Figure 4(b) to better understand the reasoning) and the entire proof. □

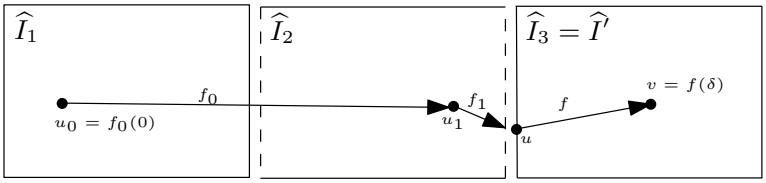
Lemma 4. *For all locations ℓ, polyhedra P and I, and $i \geq 1$, it holds*

$$W_i \subseteq ncPost_\ell(P, I, i).$$

Proof. We proceed by induction on $i \geq 1$. For the base case, let $v \in W_1$ and recalling that $W_0 = P$, by definition there exist convex polyhedra $\widehat{P}' \in [\![P]\!]$ and $\widehat{I}' \in [\![I]\!]$ such that $v \in Post_\ell\big(pentry_\ell(\widehat{P}', \widehat{I}'), \widehat{I}'\big)$. By definition of post operator, there exists a valuation $u \in pentry_\ell(\widehat{P}', \widehat{I}')$, an activity $f \in Adm(\langle l, u\rangle)$ and time $\delta \geq 0$ such that $f(\delta) = v$ and for all $0 < \delta' \leq \delta$, $f(\delta') \in \widehat{I}'$. But the last one implies $d(u, v, I) = 1$ that, together with the other conditions, allows us to write $v \in ncPost_\ell(P, I, 1)$ and this concludes the base case. For the inductive step, let $i \geq 2$ and $v \in W_i$. If the valuation v also belongs to W_{i-1}, then the thesis trivially holds for the inductive hypothesis. Otherwise, there exists convex polyhedra $\widehat{W}' \in W_{i-1}$ and $\widehat{I}' \in [\![I]\!]$ such that $v \in Post_\ell(pentry_\ell(\widehat{W}', \widehat{I}'), \widehat{I}')$ By definition of post operator, there exists a valuation $u \in pentry_\ell(\widehat{W}', \widehat{I}')$, an activity $f \in Adm(\langle l, u\rangle)$ and time $\delta \geq 0$ such that $f(\delta) = v$ and for all $0 < \delta' \leq \delta$,



(a) Case (a): $u \in \widehat{I}_{i-1} = \widehat{I}_2$. Subcase $(a1)$ occours when $\widehat{I}_1 = \widehat{I}'$ or $\widehat{I}_2 = \widehat{I}'$, otherwise we are in subcase $(a2)$.



(b) Case (b): $u \in \widehat{I}_i = \widehat{I}_3$. Subcase $(b1)$ occours when $\widehat{I}_1 = \widehat{I}'$ or $\widehat{I}_2 = \widehat{I}'$, otherwise we are in subcase $(b2)$.

Fig. 5. The cases in the proof of Lemma 4.

$f(\delta') \in \widehat{I}'$. From $u \in pentry_\ell(\widehat{W}', \widehat{I}')$ we have either (a) $u \in \widehat{W}' \cap cl(\widehat{I}')$ or (b) $u \in cl(\widehat{W}') \cap \widehat{I}'$. Case (a), that is $u \in \widehat{W}'$, trivially implies that $u \in W_{i-1}$ and then, by inductive hypothesis, $u \in ncPost_\ell(P, I, i-1)$. By definition of post operator, there exists a valuation $u_0 \in P$, an activity $f_0 \in Adm(\langle l, u_0 \rangle)$ and time $\delta_0 \geq 0$ such that $f(\delta_0) = u$, for all $0 < \delta' \leq \delta_0$, it holds $f(\delta') \in I$ and $d(u_0, u, I) = i - 1$. Now we can distinguish two subcases: $(a1)$ when the convex polyhedron $\widehat{I}'$ is one of the $i-1$ components of the invariant in which the system remains, and $(a2)$ when the system never reaches $\widehat{I}'$ during the activity f_0 that leads from u_0 to u. Figure 5(a) graphically depicts this case and the corresponding subcases. In the case $(a1)$ v belongs to $ncPost_\ell(P, I, i-1)$: one can reach v from u by concatenation f_0 and f and the system does not exit the $i-1$ (including $\widehat{I}'$) components of the invariant. So $v \in ncPost_\ell(P, I, i-1)$, which directly implies that $v \in ncPost_\ell(P, I, i)$, and we are done for the subcase $(a1)$. In the case $(a2)$, it is clear that $d(u_0, v, I) = i$ (the system reaches for the first time $\widehat{I}'$) and, recalling that $f(\delta) = v$, we can conclude that $v \in ncPost_\ell(P, I, i)$, and this conclude the subcase $(a2)$. In the case (b), we have that $u \in cl(\widehat{W}') \cap \widehat{I}'$. By definition of potential entry there exists a valuation $u_1 \in \widehat{W}'$, an activity $f_1 \in Adm(\langle l, u_1 \rangle)$ and time $\delta_1 \geq 0$ such that $f_1(\delta_1) = u$ and for all $0 \leq \delta' < \delta_1$, $f_1(\delta') \in \widehat{W}'$. But $u_1 \in \widehat{W}'$ implies that $u_1 \in W_{i-1}$ and by inductive hypothesis we can write $u_1 \in ncPost_\ell(P, I, i-1)$. Now, by definition of the post operator, there exists a valuation $u_0 \in P$, an activity $f_0 \in Adm(\langle l, u^* \rangle)$ and time $\delta_0 \geq 0$ such that $f_0(\delta_0) = u$, for all $0 < \delta' \leq \delta_0$ it holds that $f_0(\delta') \in I$ and $d(u_0, u_1, I) = i-1$. The last one means that in order to reach the valuation u_1 from the valuation u_0 (via the activity f_0) the system always remains in $i-1$ convex components of $[\![I]\!]$. Now, two different subcases can happen: $(b1)$ $\widehat{I}'$ is one of the $i-1$ visited

convex component or ($b2$) otherwise. Figure 5(b) graphically depicts this case and the corresponding subcases. Consider the subcase ($b1$) and recall that for all $0 \leq \delta' < \delta_1$, $f_1(\delta') \in \widehat{W}'$, clearly we have that $f_1(\delta') \in W_{k-1}$ and, by inductive hypothesis, $f_1(\delta') \in ncPost_\ell(P, I, i-1)$. Hence, in order to reach the valuation $u = f_1(\delta_1)$ from u_0 the system always remains in the same $i-1$ convex (included $\widehat{I}'$) and then $d(u_0, u, I) = i-1$. Then, from u to v (via the activity f) the system always remains in $\widehat{I}'$ and then $d(u_0, v, I) = i-1$. Hence, $v \in ncPost_\ell(P, I, i-1) \subseteq ncPost_\ell(P, I, i)$, and this ends subcase ($b1$). In the subcase ($b2$), we have that $d(u_0, u, I) = i-1$. Then in order to reach v from u (via activity f), the system remains in $\widehat{I}'$ (never reached before) and then $d(u_0, v, I) = i$ and hence $v \in ncPost_\ell(P, I, i)$ that ends the case and the entire proof. □

Finally, we are able to easly prove Theorem 1, by using Lemma 3 and Lemma 4 as follows:

Proof of Theorem 1 First, notice that give two valuation u and v, where $u, v \in ncPost_\ell(P, I)$, then by definition of post operator $d(u, v, I) \leq \big|[\![I]\!]\big|$. Then trivially holds that let $n = \big|[\![I]\!]\big|$, we have that $ncPost_\ell(P, I) = ncPost_\ell(P, I, n)$.

Now, by Lemmas 3 and 4, for all $k \geq n$, $ncPost_\ell(P, I) = W_k$. This means that the sequence reaches the fixed point after at most n iterations. □

Related Work In [17], the author shows a different approach in order to tackle non-convex invariants. The proposed algorithm to compute the reachable set is built only for closed convex invariants, but this is not a restriction because (closed) non-convex invariants can be modeled by splitting locations. This means that starting from an automaton A with non-convex invariants, it is necessary to build an equivalent automaton B whose locations have only convex invariants: this is done by taking, for each location of A, the exact convex covering Q of the corresponding invariant and then, for each convex component $\widehat{Q} \in Q$, by adding a location to B whose associated (convex and closed) invariant is $\widehat{Q}$. Therefore, this approach does not work with non-closed invariants and needs a postprocessing phase in order to build the automaton B. Our approach tries to overcome these limitations: the recahibility analysis is directly done by using the $ncPost_\ell$ operator, allowing the usage of non-closed invariants and avoiding the hidden process of building a new automaton.

The operator $ncPost_\ell$ has an interesting relation with a group of operators used in safety control problems. In particular with the RWA_l^{m} operator defined in [7] (also known under other names as *Reach* [25], *Unavoid_Pre* [3] and *flow_avoid* [26]). Informally, given a location l and two sets of variable valuations U and V, $RWA_l^{\mathrm{m}}(A, B)$ contains the set of valuations from which there exists an activity

that reaches A while avoiding $B \cap \overline{A}$. Formally:

$$RWA_l^{\mathrm{m}}(A,B) = \Big\{ u \in Val(X) \,\Big|\, \exists f \in Adm(\langle l, u\rangle),\, \delta \geq 0 : \\ f(\delta) \in A \text{ and } \forall\, 0 \leq \delta' < \delta : f(\delta') \in \overline{B} \cup A \Big\}.$$

Informally, we can express $ncPost_\ell(P, I)$ as the set of valuations that, starting from P, may reach I while avoiding $\overline{I}$. The main difference is given by the fact that RWA_l^{m} does not take into account a starting set of valuations, while the computation of $ncPost_\ell(P, I)$ depends on the initial set P. This is reflected also in how the operators are computed: in the case of $RWA_l^{\mathrm{m}}(A, B)$, the computation is given by a backward procedure that starts from the target set of valuations A and by taking all the valuations that avoid B. In the opposite, $ncPost_\ell(P, I)$ is computed by a forward procedure that starts from the initial set of valuations P and by taking all the valuations that avoid $\overline{I}$.

2.3 Linear Hybrid Automata with Urgency

In this section, we extend LHA with a so-called *urgency condition* each location. The urgency condition impedes time elapse, i.e., no continuous activities continue from a valuation that satisfies the condition. As we will see later, there is a connection between urgency conditions on locations and urgent semantics on transitions.

Definition and Semantics We denote by $SPoly(X)$ the subset of $\mathbb{R}^X$ that can be obtained by finite disjunction of closed convex polyhedron. A *Linear Hybrid Automaton with Urgency (LHAU)* $H = (Loc, X, Lab, Edg, Flow, Inv, Urg, Init)$ consists of a LHA defined in Sect. 2.1 and a mapping $Urg : Loc \to SPoly(X)$, called *urgency condition*. To designate the urgent states, we use the shorthand $UrgS = \bigcup_{l \in Loc} \{\ell\} \times Urg(\ell)$.

Urgent transitions In our definition, the urgency condition is defined for each location. An alternative approach, popular mainly because of its syntactical simplicity, is to designate each discrete transition as urgent or not. This is also referred to as *as-soon-as-possible (ASAP) transitions*. Urgent transitions can easily be translated to an urgency condition: Let $Edg_U \subseteq Edg$ be the set of urgent transitions. Then the equivalent urgency condition is the union of the outgoing guards,

$$Urg(\ell) = \{u \mid \exists (\ell, \eta, \ell') \in Edg_U : (u, v) \in \eta\}.$$

Semantics The urgency conditions affect only the timed steps, while the definition of discrete step remains the same as for LHA. Given a state $s = \langle l, v\rangle$, we define $loc(s) = l$ and $val(s) = v$. In order to give the semantics of timed-steps for $LHAU$ we define, for an activity $f \in Adm(s)$, the *Switching Time* of f in

l, denoted by $SwitchT(f, U)$, as the value $\delta \geq 0$ such that, for all $0 \leq \delta' < \delta$, $f(\delta') \notin U$ and $f(\delta) \in U$. When for all $\delta \geq 0$ it holds that $f(\delta) \notin U$, we write $SwitchT(f, U) = \infty$. Informally, the switching time of an activity f in the location l specifies the maximum amount of time δ such that the system, by following the activity f, is allowed to remain in the location l.

Given two states s, s', there is a *timed step* $s \xrightarrow{\delta, f} s'$ with *duration* $\delta \in \mathbb{R}^{\geq 0}$ and activity $f \in Adm(s)$ iff *(i)* there exists the timed step $s \xrightarrow{\delta, f} s'$ in the LHA without urgency conditions, and *(ii)* $\delta \leq SwitchT(f, Urg(loc(s)))$. The special timed step $s \xrightarrow{\infty, f}$, which represents the case when the system follows the activity f forever, is allowed only if, for all $0 \leq \delta' \leq \delta$, $\langle loc(s), f(\delta') \rangle \in InvS$ and $SwitchT(f, Urg(loc(s))) = \infty$.

Parallel Composition One attractive feature of urgency is that a model can be decomposed for cases where this is not possible without urgency. Consider the example of an automaton for the plant and an automaton for the controller. Without urgency, the controller automaton can in general not prevent time elapse in the plant automaton, unless an additional clock is introduced and that clock is sampled periodically, see also remarks in Sect. 1. We give a brief formal definition of parallel composition with urgency for the case where both automata range over the same variables. For a definition including input and output variables (as used in SpaceEx) see [11]. The key here is that the urgency condition of the composition is the union of the urgency conditions of the operands.

Definition 3 (Parallel composition). *Given linear hybrid automata with urgency H_1, H_2 with $H_i = (Loc_i, X, Lab_i, Edg_i, Flow_i, Inv_i, Urg_i, Init_i)$, their* parallel composition *is the LHAU $H = (Loc_1 \times Loc_2, X, Lab_1 \cup Lab_2, Edg, Flow, Inv, Urg, Init)$, written as $H = H_1 \| H_2$, where*

- $((l_1, l_2), \alpha, \eta, (l_2', l_2')) \in Edg$ *iff*
  - $\alpha \in Lab_1 \cap Lab_2$, *for* $i = 1, 2$, $(l_i, \alpha, \eta_i, l_i') \in Edg_i$, *with* $\eta = \eta_1 \cap \eta_2$, *or*
  - $\alpha \notin Lab_1$, $l_2' = l_2$, *and* $(l_1, \alpha, \eta, l_1') \in Edg_1$, *or*
  - $\alpha \notin Lab_2$, $l_1' = l_1$, *and* $(l_2, \alpha, \eta, l_2') \in Edg_2$;
- $Flow(l_1, l_2) = Flow_1(l_1) \cap Flow_2(l_2)$;
- $Inv(l_1, l_2) = Inv_1(l_1) \cap Inv_2(l_2)$;
- $Urg(l_1, l_2) = Urg_1(l_1) \cup Urg_2(l_2)$;
- $Init(l_1, l_2) = Init_1(l_1) \cap Init_2(l_2)$.

Reachability The discrete post operator for the class of $LHAU$ is trivially the same of the classical one, while the continuous one, that we call *Urgent Continuous Post Operator*, changes due to the extra condition induced by the operator $SwitchT$:

Definition 4 (Urgent continous post). *Given a linear hybrid automaton with urgency H, a location $\ell \in Loc$, and a set of valuations $P \subseteq Inv(\ell)$, let $I =$*

$Inv(\ell)$, and $U = Urg(\ell)$. The urgent continuous post operator *$UPost(P, I, U)$ is defined as:*

$$UPost(P,I,U) = \Big\{ v \in val(X) \Big| \exists u \in P, f \in Adm(\langle \ell, u \rangle), \delta \geq 0 : \\ f(\delta) = v, \text{ for all } 0 < \delta' \leq \delta, f(\delta') \in I, \text{ and } \delta \leq SwitchT(f,U) \Big\}.$$

Notice that, by definition, $UPost_\ell(P, I, U) \subseteq ncPost_\ell(P, I)$.

2.4 Computing the Urgent Continuous Post Operator

We now derive a construction of the urgent post operator, starting with the post operator for non-convex invariants and adding the states that are missing.

The urgent post operator has to compute the valuations that are reachable from some set P without passing through states in the urgent set U. This includes the states that are reachable within the complement of U, so $ncPost_\ell(P \cap \overline{U}, I \cap \overline{U})$ is an underapproximation of $UPost_\ell(P, I, U)$. In the following, let $\mathbb{V}_{nc} = ncPost_\ell(P \cap \overline{U}, \overline{U})$ and $\mathbb{V}_U = UPost_\ell(P, I, U)$. The set $\mathbb{V}_{nc}$ trivially does not contain valuations that belong to U (by definition of invariant), while $\mathbb{V}_U$ also contains those valuations that touch U for the first time on a run. Indeed, while the system is allowed to remain on the boundary of an invariant for any time as the invariant is satisfied, the system can not remain on the boundary of an urgency condition because in the instant the urgency condition is meet, the system can not evolve any more, i.e., it is forced either to stop the evolution of the continuous variables or to jump in another location. Figure 6 illustrates the relationship between $ncPost_\ell(P \cap \overline{U}, I \cap \overline{U})$ (left column of the figure) and $UPost_\ell(P, I, U)$ (right column of the figure) with several examples.

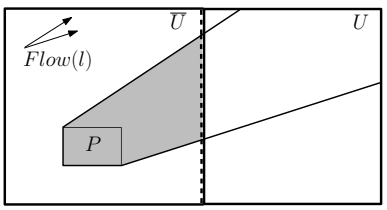
We compute $\mathbb{V}_U$ by adding (i) P itself (ii) the reachable states in $\bar{U}$ $\mathbb{V}_{nc}$, and (iii) the valuations that belong to the boundary between $\mathbb{V}_{nc}$ and U from where it is possible to reach U by following some admissible activity. This is formalized as follows:

Theorem 2. *Given a location $\ell \in Loc$ and a set $P \subseteq Inv(\ell)$, let $I = Inv(\ell)$, $U = Urg(\ell)$, $\mathbb{V}_{nc} = ncPost_\ell(P \cap \overline{U}, I \cap \overline{U})$, and $B = \bigcup_{\widehat{A'} \in [\![\mathbb{V}_{nc}]\!]} \bigcup_{\widehat{U'} \in [\![U]\!]} pentry_\ell(\widehat{A'}, \widehat{U'} \cap I)$. Then*

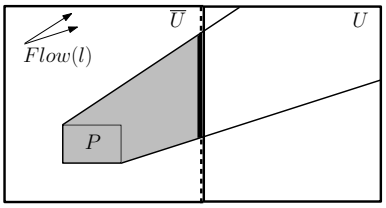
$$UPost_\ell(P, I, U) = P \cup \mathbb{V}_{nc} \cup B.$$

Proof. $[\subseteq]$ Let $v \in UPost_\ell(P, I, U)$, by definition there exists a valuation $u \in P$, an admissible activity $f \in Adm(\langle \ell, u \rangle)$ and a real value $\delta \geq 0$ such that $f(\delta) = v$, for all $0 < \delta' \leq \delta$ it holds that $f(\delta') \in I$, and $\delta \leq SwitchT(f, U)$. Moreover, due to the fact that $P \subseteq I$, we have also that $f(0) \in I$ and then, for all $0 \leq \delta' \leq \delta$, we have that $f(\delta') \in I$. Now, we can distinguish three different cases on $SwitchT(f, U)$:

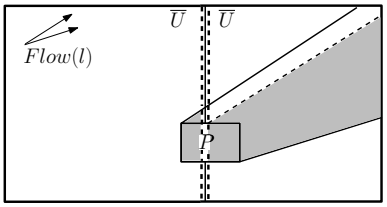
1. If $SwitchT(f, U) = 0$, then $\delta = 0$. By definition of $SwitchT$, we have that $u = v = f(0) \in U$, and then $v \in (P \cap U)$, and the thesis holds.



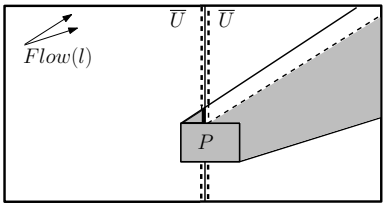
(a) $\mathbb{V}_{nc}$ with $U = \{x \geq d\}$.



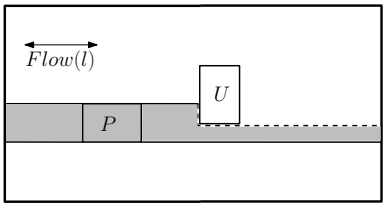
(b) $\mathbb{V}_U$ contains the boundary of $\mathbb{V}_{nc}$ with U.



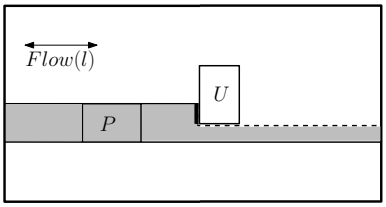
(c) $\mathbb{V}_{nc}$ with $U = \{x = d\}$.



(d) $\mathbb{V}_U$ contains $P \cap U$ and the boundary of $\mathbb{V}_{nc}$ with U.



(e) $\mathbb{V}_{nc}$ with flow tangential to U.



(f) $\mathbb{V}_U$ contains only the reachable part of the boundary of $\mathbb{V}_{nc}$ with U.

Fig. 6. The urgent post states $\mathbb{V}_U = \mathit{UPost}_\ell(P, I, U)$ can be obtained from $\mathbb{V}_{nc} = \mathit{ncPost}_\ell(P \cap \overline{U}, I \cap \overline{U})$ plus a part of the boundary between $\mathit{ncPost}_\ell(P \cap \overline{U}, I \cap \overline{U})$ and U. Here, the invariant I is defined to be *true*. The dashed lines identify the non-closed borders.

2. If $0 \neq \delta < \mathit{SwitchT}(f, U)$, then for all $0 \leq \delta' \leq \delta$ it holds $f(\delta') \in \overline{U}$. The latter, coupled with the conditions on the invariant, gives us $f(\delta') \in I \cap \overline{U}$, and then $v \in A$.
3. If $0 \neq \delta = \mathit{SwitchT}(f, U)$ then, for all $0 \leq \delta' < \delta$ it holds that $f(\delta') \in \overline{U}$, and $f(\delta) \in U$. Moreover, for all $0 \leq \delta' < \delta$, $f(\delta')$ also belongs to $\mathbb{V}_{nc}$ ($f(\delta')$ is always in the invariant $I \cap \overline{U}$). This means that, starting from a valuation in $\widehat{A}' \in [\![\mathbb{V}_{nc}]\!]$ it is possible to reach $f(\delta)$ in $\widehat{U}' \in [\![U]\!]$, and by definition of potential entry we have that $v = f(\delta) \in \mathit{pentry}_\ell(\widehat{A}', \widehat{U}' \cap I) = B$.

[$\supseteq$] Considering the valuation v that belongs to the r.h.s. of the formula in Theorem 2, we can distinguish three cases, based on at what disjunct among A, B or $(P \cap U)$ the valuation v belongs.

1. If $v \in (P \cap U)$, then there exists an activity $f \in Adm(\langle \ell, v \rangle)$ and $\delta = 0$ such that $v = f(\delta) \in P \cap I \cap U$. Hence $f(0)$ is in the invariant I and $SwitchT(f, U) = 0$ and we can conclude that $v \in UPost_\ell(P, I, U)$.
2. If $v \in A$, then by definition of $ncPost_\ell(P \cap \overline{U}, I \cap \overline{U})$, there exists a valuation $u \in P \cap \overline{U}$, an activity $f \in Adm(\langle \ell, u \rangle)$ and time $\delta \geq 0$ such that, for all $0 \leq \delta' \leq \delta$, we have $f(\delta') \in I \cap \overline{U}$. This clearly implies that $SwitchT(f, U) > \delta$, and we can conclude that $v \in UPost_\ell(P, I, U)$.
3. If $v \in B$ then, by definition of B, there exists $\widehat{A}' \in [\![\mathbb{V}_{nc}]\!]$ and $\widehat{U}' \in [\![U \cap I]\!]$ such that $v \in pentry_\ell(\widehat{A}', \widehat{U}' \cap I)$. Using the potential entry definition, we have that there exists a valuation $u \in \widehat{A}'$, an activity $f \in Adm(\langle \ell, u \rangle)$ and a time $\delta \geq 0$ such that $v = f(\delta) \in bndry(\widehat{A}', \widehat{U}' \cap I)$ and for all $0 \leq \delta' < \delta$, it holds that $f(\delta') \in \widehat{A}'$. The last one implies, by definition of $\mathbb{V}_{nc}$ (i.e. post operator), that (i) for all $0 \leq \delta' < \delta$, $f(\delta') \in I \cap \overline{U}$. Moreover, the fact that $U \in SPoly(X)$ and $f(\delta) \in bndry(\widehat{A}', \widehat{U}' \cap I)$ implies that $f(\delta) \in cl(\widehat{A}') \cap \widehat{U}' \cap I$ and then (ii) $f(\delta) \in \widehat{U}' \cap I$. By coupling conditions (i) and (ii) we have that (iii) $\delta = SwitchT(f, U)$. But conditions (i), (ii) and (iii) are the definition of the urgent post operator, allowing us to write $v \in UPost_\ell(P, I, U)$.

□

Related Work A general class of hybrid automata with urgency conditions is described in detail in [23], but without giving a way to compute the continuous post operator that takes into account the extra constraints coming from the urgency. In that work, the authors define the *Time Can Progress (tcp)* predicate that roughly speaking specifies, for each location $l \in Loc$, the maximum sojourn time at l, which may depend on the values of the variables when entering the location. There are two main differences between our urgency condition and the *tcp* predicate. First of all, an urgency condition on location l defines when the system is constraint to exit from l, while *tcp* describes exactly the opposite. Then, informally speaking, we can say that in order to model, by using our formalism, a *tcp* predicate associated to a location l, we need to attach to l an urgency condition that is the complement of *tcp*. In [23] the authors also describe several kind of policies on urgency. One of them, called *Synchronous Scheduling Policy*, exactly describes our approach: the location must be left as soon as possible an edge becomes enabled (the urgency condition is meet, in our case). This policy is realized by setting the *tcp* predicate equal to the complement of the disjunction of the outgoing guards from the related location. This means that, by using our urgency condition, in order to model multi-outgoing asap transition from a location ℓ, we have to define $Urg(l)$ as the disjunction of the outgoing guards transitions. This give us another motivation to choose non-convex polyhedra to define the urgency condition: by this way, we are able to easily model a system with multiple outgoing transition with asap guards. Notice that the semantics specified in [23] requires that in the exact moment that a location ℓ is entered, the *tcp* associated to ℓ must be satisfied. In our framework we relax this constraint, by allowing to jump in ℓ even if its urgency condition is already satisfied: in this

case, the system must exit from ℓ instantaneously. The rational behind this choice coming from the usual asap transitions semantics: considering a location ℓ_1 with an outgoing transition that must be taken in the exact moment that a variable assumes a fixed value. Even if the target location ℓ_2 has another asap outgoing transition whose guard is already satisfied (and hence the system is constrained to exit from ℓ_2 without time progress), the system is still allowed to jump to ℓ_2 and escape from it instantaneously. For this motivation, our $UPost^c_{\ell_2}(P, I, U)$ operator also contains all the valuations that at entering time (valuations that belong to P) already meet the urgency condition.

Our definition of urgency condition is closer to the *stopping condition* defined in [14] (in the context of timed automata), that is a predicate on the clock-variables of an automaton, associated to each location, which allows passing of time in the location as long as the stopping condition is false. The urgency in hybrid automata is also described in the *Computational Interchange Format for Hybrid Systems (CIF)* (see [5]), accepted as standard for modelling hybrid systems. As in our case, the time is allowed to progress until the urgency condition is false. The difference is that, similarly to the *tcp* predicate, *CIF* requires that in the exact moment that a location is entered, the valuation of the variables have to satisfies the condition.

Urgent locations In the classic LHA model checker HyTech, a transition can be designated as urgent by adding the keyword ASAP [17]. But this is restricted to transitions without guard constraints locally, as well as in the composed model (see [15, 17] for more details). This is equivalent to having *urgent locations*, i.e., locations in which time progress is not allowed. The real-time verification tool UPPAAL [6] similarly features urgent locations and urgent channels (synchronization labels) that can be used only on transitions without guard constraints. Urgent locations are semantically equivalent to adding an extra variable t, with dynamics $\dot{t} = 1$, that is set to zero when the location is entered and by attaching the invariant $t = 0$ to the location. It is easy to check that this technique can not be applied if we want to model even just a slightly more complex urgency condition. For example, consider a system with a variable x and a location, say ℓ_1, where the derivative of x can be zero, and the location must be left when $x = 0$. There is no way to model this behaviour just using invariants: by defining $(x < 0) \vee (x > 0)$ as invariant, the system can never reach the state $x = 0$. In the other hand, by relaxing the invariant in order to allow the system to reach a state with $x = 0$, the system is allowed to remain in location ℓ_1 always.

In previous versions of our model checker PHAVer, transitions could be designated as urgent, but only if the guard consists of a single constraint, locally as well as in the composed model [13]. This restriction was imposed because it suffices to be able to compute the urgent post using the standard post operator for convex invariants.

Almost ASAP In [27] the authors propose a relaxed semantics on asap transition in the context of the timed automaton, for the so called *almost asap* by delay δ. In practice, they define the *guard enlargement*, that means that transitions

can be taken also with δ time delay. The rationale behind this approach is that no hardware can guarantee that a transition will always be taken in the exact moment as defined in theory. We could define a similar approach, not only on clock variables, in a simple but opposite way: it is enough to define the urgency condition by narrowing all the constraints by a quantity that is equal to the maximum variation of the variable in the time δ.

2.5 Examples

The following examples shall illustrate the application of the non-convex and urgent post operators. The first example is a set of test cases for the implementation. The second example is a small case study modeling a batch reactor system.

Test Cases Figure 7 depicts two test cases. The first one (Figure 7(a)) is used to check cases (a) and (b) of Lemma 3: the two cases are verified because one time there is a potential entry all inside $\widehat{I}_2$ (by going from $\widehat{I}_1$ to $\widehat{I}_2$), and one time there is a potential entry that is not inside $\widehat{I}_3$ (by going from $\widehat{I}_2$ to $\widehat{I}_3$). Notice that in the situation shown in Figure 7(a), all the founded potential entry really are entries. This test case is described as follows:

- Init set $P \equiv x \geq 1$ *and* $x \leq 2$ *and* $y \geq 3$ *and* $y \leq 6$;
- Invariant $Inv = \widehat{I}_1 \cup \widehat{I}_2 \cup \widehat{I}_3 \cup \widehat{I}_4$;
- $\widehat{I}_1 \equiv 0 \leq x < 3$ *and* $0 \leq y \leq 8$;
- $\widehat{I}_2 \equiv 3 \leq x \leq 5$ *and* $1 \leq y \leq 4$;
- $\widehat{I}_3 \equiv 5 < x \leq 8$ *and* $0 \leq y \leq 4$;
- $\widehat{I}_4 \equiv 6 \leq x \leq 9$ *and* $4 < y \leq 7$;
- Flow $F \equiv \dot{x} = 1$ *and* $-0.5 \leq \dot{y} \leq 0.5$.

The set of all the reachable valuations is described as follows:

$$\big(1 \leq x < 3 \text{ and } -0.5 \cdot x + 3.5 \leq y \leq 0.5 \cdot x + 5.5\big)||$$

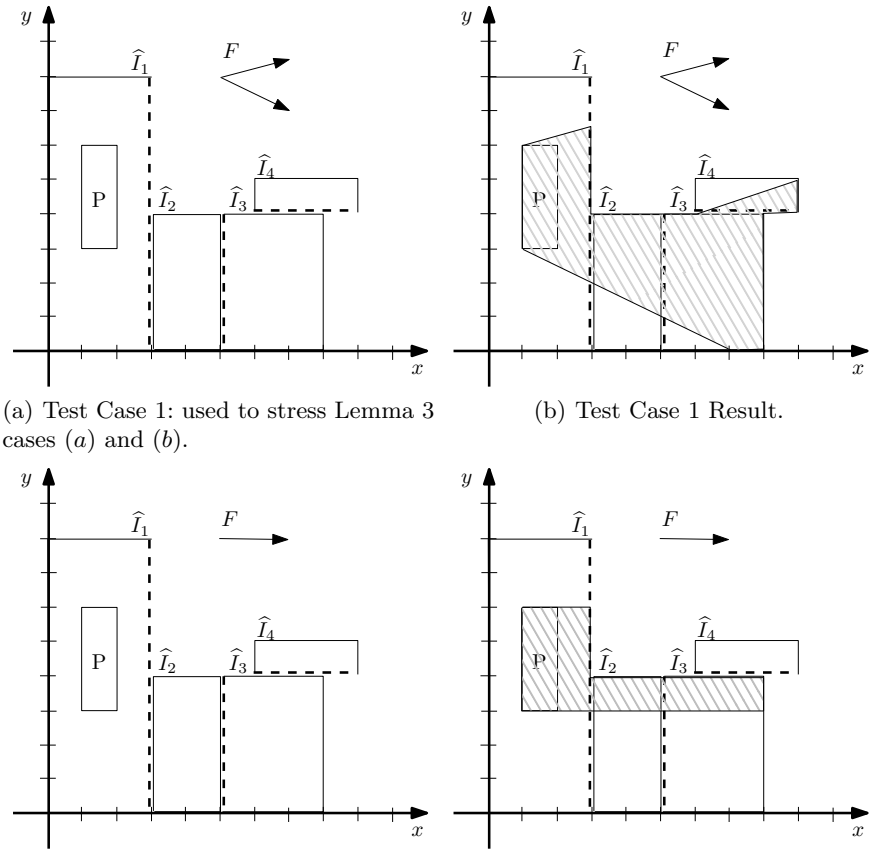
$$\big(3 \leq x \leq 5 \text{ and } -0.5 \cdot x + 3.5 \leq y \leq 4\big)||$$

$$\big(5 < x \leq 8 \text{ and } y \geq -0.5 \cdot x + 3.5 \text{ and } 0 \leq y \leq 4\big)||$$

$$\big(6 < x \leq 9 \text{ and } 4 < y \leq 0.5 \cdot x + 1\big).$$

The second test case (Figure 7(c)) is built with the same invariant shape of the first one (and then it is usuful to stress the two cases of Lemma 3) but the real aim here is to see what happens when there is a non-empty potential entry that actually it is not a real entry: here, due to the nature of the flow (it is constant and described by $\dot{x} = 1$, that is no vertical movement is allowed) when the system reach the convex component $\widehat{I}_3$, even if it is possible to identify a potential entry to $\widehat{I}_4$, the flow is such that it is impossible to enter in $\widehat{I}_4$ from $\widehat{I}_3$.

This test case is described as follows:



(a) Test Case 1: used to stress Lemma 3 cases (a) and (b).

(b) Test Case 1 Result.

(c) Test Case 2: used to stress the potential entry definition.

(d) Test Case 2 Result.

Fig. 7. Two Test Cases with the results.

- Init set $P \equiv x \geq 1$ *and* $x \leq 2$ *and* $y \geq 3$ *and* $y \leq 6$;
- Invariant $Inv = \widehat{I}_1 \cup \widehat{I}_2 \cup \widehat{I}_3 \cup \widehat{I}_4$;
- $\widehat{I}_1 \equiv 0 \leq x < 3$ *and* $0 \leq y \leq 8$;
- $\widehat{I}_2 \equiv 3 \leq x \leq 5$ *and* $1 \leq y \leq 4$;
- $\widehat{I}_3 \equiv 5 < x \leq 8$ *and* $0 \leq y \leq 4$;
- $\widehat{I}_4 \equiv 6 \leq x \leq 9$ *and* $4 < y \leq 7$;
- Flow $F \equiv \dot{x} = 1$ *and* $\dot{y} = 0$.

The set of all the reachable valuations is described as follows:

$$\big(1 \leq x < 3 \text{ and } 3 \leq y \leq 6\big)||$$

$$\big(3 \leq x \leq 5 \text{ and } 3 \leq y \leq 4\big)||$$

$$\big(5 < x \leq 8 \text{ and } 3 \leq y \leq 4\big)||.$$

Batch Reactor To showcase the algorithm an its implementation, we present a modular model of a batch-reactor system, which is a variation of the case study in [4]. It shall illustrate the following points:

- urgency conditions simplify the modeling process greatly because models can be better decomposed and each module can be simpler;
- urgent and non-urgent transitions arise naturally in practice, as abstractions that reflect the degree of determinism and knowledge about the system;
- urgent transitions with more than one guard constraint also arise naturally, illustrating how limiting the restrictions of HyTech and PHAVer can be;
- the requirement that urgency conditions be closed is of lesser importance in practice, since individual guard constraints need not be closed.

The batch reactor is comprised of a reactor R1 and two buffer tanks B2,B3 connected by pipes. The reactor is used to create a product that is then made available to a consumer in the two buffer tanks, see the schematic in Fig. 8(a). A controller measures the fill levels in the reactor and the buffers, and opens and closes valves connecting the reactor to the buffers in order to produce and deliver the product to the consumer. The specification is to verify that neither buffer ever becomes empty, and that none of the tanks overflows.

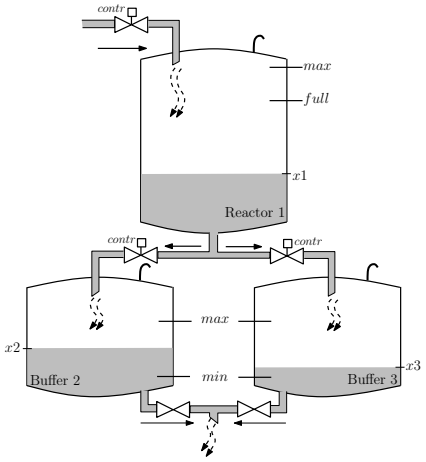
We now present the LHA models.

Controller The controller automaton is shown in Fig. 8(d). The opening and closing of valves is modeled by synchronization labels. In the production step, the reactor is filled with educts (raw materials) coming from the outside. Details on the filling and reaction process itself are omitted since they are irrelevant to this example, but it does take a certain amount of time and produces an uncertain amount of product. This is modeled by the fact that the controller ends the filling process when the reactor level $x_1 \in [x_{1,full}, x_{1,max}]$, which is accomplished with the invariant $x_1 \leq x_{1,max}$ and a non-urgent transition with label *close_in* and guard condition $x_1 \geq x_{1,full}$. When the product is ready, the controller decides whether to fill buffer B2, buffer B3, or wait. The controller decides which buffer to fill using the following simple criteria:

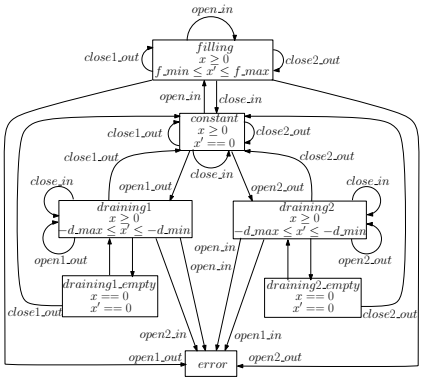
- To avoid overflow, never start filling a buffer above a given maximum level.
- To avoid empty buffers, fill a buffer below a given minimum level.
- If the above is met, fill the buffer with the lower level.
- To be deterministic, prioritize B2.

All transitions for filling buffers B2 and B3 are urgent. The if-then-else structure of the criteria leads to guards with more than one constraint, some of which are strict inequalities. Thanks to the urgency, the controller model requires no clocks or while-loops.

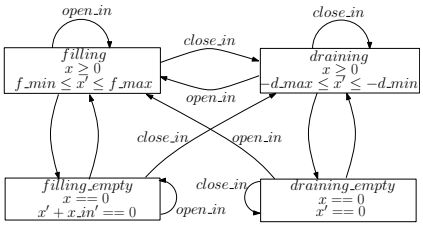
Reactor The reactor automaton is shown in Fig. 8(b). The locations of the reactor correspond to the different combinations of open and closed valves. If the tank is empty, the dynamics of the locations $draining_i$ lead to a deadlock, so



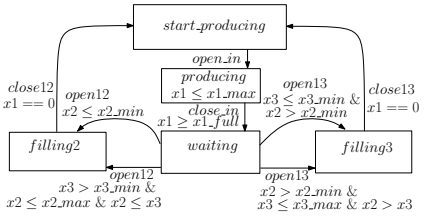
(a) Schematic of the Batch-Reactor System.



(b) Automaton Model of the Reactor.



(c) Automaton Model of the Buffers.

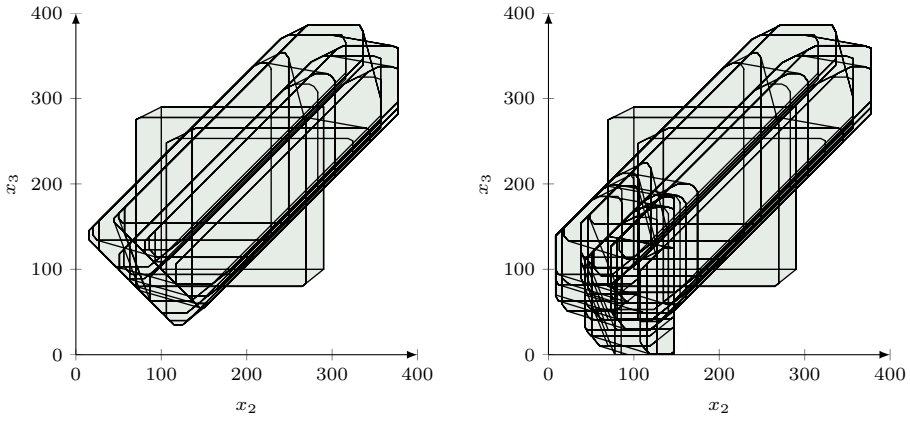


(d) Automaton Model of the Controller.

Fig. 8. Batch Reactor System: Schematic and Automata Models.

locations $draining_i\_empty$ with $\dot{x}_1 = 0$ are included to allow time to elapse. The model is simplified using the assumptions that the reactor must not be filled and drained at the same time (a common requirement in chemical engineering), and that only one of the buffers is filled at any given time. An error location is included so that violations of these assumptions can be detected. The transitions are not set as urgent in this automaton; the urgency in the composed system results from the controller.

Buffers The buffers are modeled each as an instantiation of the automaton shown in Fig 8(c). The outflow of the buffers is determined by the consumer, and therefore only known within the bounds $[-d_{i,max}, -d_{i,min}]$ (there is no valve to control outflow). The inflow is determined is equal to the outflow of the reactor. This leads to the dynamics $\dot{x}_i = [-d_{i,max}, -d_{i,min}] - \dot{x}_1$. Note that $\dot{x}_1$ is negative when the buffer is filling, so $\dot{x}_i$ is augmented by $-\dot{x}_1$ in this dynamics. Again, the transitions are not set as urgent; the urgency in the composed system results from the controller.



(a) reachable buffer levels x_2 and x_3 for safe parameter values

(b) decreasing the min. reactor outflow by 5% eventually leads to an empty buffer B3

Fig. 9. The evolution of the continuous variables of the batch reactor example, starting from location *start_producing* with initial values $x_1 = 0$, $x_2 = 100$ and $x_2 = 100$

The specification was verified using SpaceEx/PHAVer (an implementation of the PHAVer reachability algorithm built on the SpaceEx platform). The inflow and outflow rates were set nondeterministically to be within intervals; the models incl. parameter values are available at `spaceex.imag.fr`. The computation of the complete reachable states shown in Fig. 9(a) takes 3.0 s and 24 MB of memory on a standard laptop. Finding the fixed point takes a total of 178 continuous post operations.

2.6 Conclusions

Linear Hybrid Automata stand out in the hybrid systems domain because sets of successor states can be carried out exactly. Available algorithms are require convex invariants and single-constraint urgency conditions. In this work we propose algorithms that can handle non convex invariants and (closed) non-convex urgency conditions. The practical impact is that this extension can be used in order to model system in which transitions have to be taken as soon as possible. This is a common feature in several commercial tools used as de-facto standard in the industry (for example in the automotive context) such as Matlab/Simulink or Modelica. The proposed algorithms overcome some limitations of previous attempts, as the necessity to use only closed invariants and direct works on the original automaton, without postprocessing phase on it. We formally proved the correctness and the termination of the proposed procedures, which are based on two operators: the first one is the classical continuous post operator $Post_\ell$ and the other one (defined here) is the so-called potential entry operator $pentry_\ell$. To the best of our knowledge, the proposed solutions represent the first sound and complete procedures for the task in the literature. We extended the tool PHAVer with our procedures and illustrated the results on an example.

The results from this paper will serve as the basis for future work on hybrid automata with more complex (piecewise affine and nonlinear) dynamics. The approach will need to be adapted since the specialized successor computations for these classes, e.g., in [12], only handle closed sets. The results shown in this paper could be used as base for some interesting future works: for example, we are currently planning to build a tool in order to automatically convert a subset of Simulink diagrams in an equivalent LHA. This became possible because the asap transitions can be exactly translated by using the urgency conditions. The proposed algorithms can be than used on the obtained automaton to compute the set of reachable states. This may improve the capability to discover errors during the design stage, because Simulink is only able to perform simulation. In a similar way, we can build an automatic translator even for other industrial tools, as Modelica.

3 Simulink

In this Section we introduce and then we try to formally describe the modeling language of Simulink. Simulink is an environment for simulation and model-based design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing. The modeling language lacks a formal and rigourous definition of its semantics. Hence, here we present our estimate of the semantics then used to build out trnaslator.

3.1 Simulink Models

A Simulink design is represented graphically as a diagram consisting of interconnected Simulink blocks. It represents the time-dependent mathematical relationship betweem the inputs, states, and outputs of the design.

A Simulink model is a tuple $SL = \langle D, B, C, StartTime, StopTime \rangle$, where:

- D is a finite set of the typed variables, partitioned into the set of input variables D_i and the set of the output variables D_o;
- B is a finite set of *Simulink Blocks*. Each block $b \in B$ has input variables, output variables and some parameters P, whose completely define the nature of the block. The input and the output variables are associated to the corresponding input and output block ports, and our convention here is to identify the i-th input variable by the name In_i, and the j-th output variable by Out_j. A block can be itself a Simulink Diagram (a *SubSystem* block type);
- $C \subseteq (B, OutPorts(B)) \times (B, InPorts(B))$, where $OutPort : B \to Ports$ (resp., $InPort : B \to Ports$) is a relation that associates to each block $b \in B$ the set of his inports (resp., outports), identifies connections among blocks.

A connection $c = \langle (b, out), (b', in) \rangle \in C$, where $b, b' \in B$, $out \in OutPort(b)$ and $in \in InPort(b)$, connectes the outport out of b to the inport in of b', and rapresents a data flow between the corresponding variable belonging to b and b'.
- $StartTime \in \mathbb{R}^{\geq 0}$ is the start-time simulation;
- $StopTime \in \mathbb{R}^{\geq 0}$ is the stop-time simulation.

3.2 Simulink Blocks and Blocks Parameters.

Simulink provides a very huge number of different blocks. They are completely described by the follows parameters:

- **Name**. Identifies the name of the block. It is unique for each blocks belonging to the same subsystem;
- **BlockType**. This parameters identifies the type of the block, i.e. the type of the operation that it performs. Some of them are the following
  - *Inport*: used to model an entry point to the Simulink model or to a SubSystems BlockType. Clearly, it has no input variables and a single output variable;
  - *Outport*. Similar to the former, used as exit point with no output variables and a single input variable;
  - *SubSystem*: is a composed-block, used to model a Simulink diagram in order to allows the use of the hierarchy. Clearly, the number of the input and output variables of this blocks depends on the number of the inport and the outport associated to the modeled Simulink diagram;
  - *Constant*: used to model a constant value. This block has no input variable and a single output variable that provide the constant value associated to the block;
  - *Gain*: used to apply a gain from an input value. The block has a single input variable and a single output variable;
  - *Sum*: used to model the arithmetic sum among the input variable. The number of the input and the sequence of the operation can be choose by the user;
  - *Product*: used to model the product operation. Similar to the Sum BlockType, it is possible to choose the number of the input and the sequence of the operation (a pure product or a division);
  - *Integrator*: used to integrate the value in input. The basic Integrator block has one input variable, one output variable and also an internal state. It is also possible to choose the Reset version, where depending on the value of a secondo input variable, the internal state is reset;
  - *Switch*: used to model a change of the discrete state of the system. Usually it consists on the three input variables in_1, in_2 and in_3 and the value of the output variable *out* is equal to in_1 or in_3 depends on some conditions expressed on the value of in_2. These conditions are described by the parameters *Criteria* and *Treshold*. The first one may be $\geq Treshold$, $> Treshold$ or $\neq Treshold$, while the second one is a Real value. The

output function is given by $out = in_1$ if in_2 satisfy Criteria and Treshold, $out = in_3$ otherwise.
  - *Goto* and *From*: just used to simplify the graphical representation, in order to connect blocks without a track an hard line;
  - *MultiPortSwitch*: is a more complex Switch, with more input variables and with a more complex behaviour;
  - *DeadZone*: usually has a single input variable and a single output variable, used to output the zero value when the input is between a specified range;
  - *Scope*: used in order to monitor the values provided by a block. It is possibile to define an arbitrarly number of input variables (each of them connected to the output of a block, in order to watch the output value) and it has no output variables;
  - *Mux*, *Demux*, *DeadZone*, ...
- **Ports**. This parameter is used in order to specify the number of the input and the output variables;
- **Position**. It is a graphical parameter, used to specify the position and the dimension of the block;
- **Value**. Used with Constant blocktype, to specify the constant value associated to the block;
- **Gain**. Defined together with a Gain blocktype, specifies the gain value associated with the block;
- **Inputs**. Commonly used together with the Sum or the Product blocktype, in order to define the sequence of the operation among the input variables. Sometimes is also used with other blocks (i.e. MultiPortSwitch, Mux) in order to define the number of input variables;
- **Outputs**. Used to define the number of the output variables, in the case of some blocktype like Demux;
- **Criteria** and **Treshold**. These parameters are specified inside the definition of a Switch blocktype, in order to define the criteria and the threshold of the Switch, i.e. the rule that governs the change of the discrete state modeled by the Switch;
- **ExternalReset**. Inside the definition of an Integrator, when it is specify means that the internal state of the Integrator can be reset when a secondo input go up or down zero (resp. Rising or Falling Reset);
- **GotoTag**. Used together with Goto and From blocktype, to specifies a "wireless" connection.

Example 3. Considering a Simulink Block defined by the following parameters:

- $Name = Add$;
- $BlockType = Sum$;
- $Position = [10, 10, 30, 30]$;
- $Ports = [3, 1]$;
- $Inputs = ++-$.

$$save\_system('my\_Diagram','my\_XMLDiagram.xml','ExportToXML', true)$$

Fig. 10. Command Line to Convert .mdl native file to XML representation.

This means that the block *Add* is a *Sum* blocktype, graphically placed in the position $10, 10$ (upper-left corner) and $30, 20$ (lower-right corner). Notice that the position of the two corners also define the width and the height of the graphical representation of the block. The value associated to the parameter *Ports* specifies three input variables (In_1, In_2 and In_3), and one output variable (Out_1), while the value of the *Inputs* means that the output function is given by

$$Out_1 = +In_1 + In_2 - In_3.$$

Example 4. Similarly to the Example 3, considering a Simulink Block with the parameters:

- $Name = ControlMode$;
- $BlockType = Switch$:
- $Position = [50, 40, 70, 80]$;
- $Criteria = u2 \geq Threshold$;
- $Threshold = 10$.

These parameters define a block called "ControlMode", that is a Switch whose position/dimension is identified by *Position* (in the same way of Example 3). Been a Switch, there are three input variables (In_1, In_2 and In_3) and a single output variable (Out_1), whose value is set equal to In_1 or $In3$ depending on the value of the input variable In_2 w.r.t the *Criteria* and the *Threshold* parameters, in this way:

$$Out_1 = \begin{cases} In_1 & \text{if } In_2 \geq 10, \\ In_3 & \text{otherwise.} \end{cases}$$

3.3 The Representation of Simulink Diagram in XML File Format.

Usually a Simulink Diagram is saved on a .mdl file, whose format is not a standard. Luckily the Mathwork tools provide a useful function in order to translate the .mdl file in XML format, that is a best starting point for a translation. The function is $save\_system(String, String, String, boolean)$ and for example, starting from a Simulink Model stored on the file called *my_Diagram.mdl*, it is possible to obtain the corresponding file *my_XMLDiagram.xml* in XML format by using, on the Matlab Command Window, the command shown in Figure 10.

<?xml version="1.0" encoding="UTF-8"?>.

Fig. 11. XML Header.

The obtaint XML format file will be the input file for our tool *Simulink to SpaceEX Automatic Translator (SL2SX)*. Before to start to describe how the translator is implemented, in this Section is shown how the model is represented by XML format (Here only the meaningfull (for this work) features are shown).

The first row is used just to define the xml version and the encoding, as Figure 11 shows.

The diagram description starts by the tag *Model* (see Figure 12), that also contains the name of the higher-level System name. This value will be used in order to automatically define both the file name and the higher-level Network Component of the generating SpaceEx .xml file.

```
<ModelInformation Version="0.9">
   <Model Name="my_Diagram">
      ...
   < \Model>
< \ModelInformation>
```

Fig. 12. Specification of the Diagram Name.

Two important parameters that define the Simulink Simulation are given under the *Configuration Set*, as you can see in Figure 13: the Simulation Start and Stop time are defined by the corresponding tag under the earlier mentioned section. This values will be usuful in order to write the configuration SpaceEx file, where the simulation parameters are defined.

Before the real definition of the Simulink diagram, the XML descrition specifies the default parameters values associate to each block present in the whole diagram: this mean that, in the definition of the system, when a block is involved without any parameter, the corresponding value is taken from this section. This allows to fully describe the block. Clearly here one can find, for each block, all the possible parameters associate to it described above (*Value*, *Inputs*, *Outputs*, *Ports*, *Criteria*, *Threshold*, etc.). Considering the example depicted in Figure 14, if in the definition of the system there is a Constant Block without the specification of the parameter (*Value*), the block is automatically associated to the value 4.

Once defined the default parameters for the blocks, is the time to define the effective the Simulink diagram composition. In the XML format the begin of the system specification is given by the tag *System*, as you can see in Figure 15.

The composition of the diagram is give as a list of blocks. Recall the definition above, a block can be *basic* or a *SubSystem*. Figure 16 shown the definition of a basic block. In details, a Switch block type identified by the name *ControlMode*

```
<ModelInformation Version="0.9">
  <Model Name="my_Diagram">
    ...
      <ConfigurationSet>
        <Array PropName="ConfigurationSets" ... >
          <Object Version="1.12.1" ... >
            <Array PropName="Components" ... >
              <Object Version="1.12.1" ... >
                <P Name="StartTime"> 0.0 </P>
                <P Name="StopTime"> 50.0 </P>
                ...
              < \Object>
            < \Array>
          < \Object>
        < \Array>
      < \ConfigurationSet>
    ...
  < \Model>
< \ModelInformation>
```

Fig. 13. Specification of Simulation Start and Stop Time.

```
<ModelInformation Version="0.9">
  <Model Name="my_Diagram">
    ...
      <BlockParameterDefaults>
        <Block BlockType="Constant">
          <P Name="Value"> 4 </P>
          ...
        < \Block>
        ...
      < \BlockParameterDefaults>
    ...
  < \Model>
< \ModelInformation>
```

Fig. 14. Specification of the Default Values of the Blocks Parameters.

is defined, whose parameters *Criteria* and *Threshoild* are explicitly defined and the block is completely described. Notice that, if the last parameters was not specified, the parameters would come from the default section.

In order to allow hierarchy, Simulink provide the possibility to define a Block as a *SubSystem*. In this way is possibile to define a Simulink diagram into an higher level Simulink diagram. Figure 17 show the XML representation of the subsystem definition. In particolar, the subsystem *Force* is desribed. The graphical position is $[545, 557, 620, 653]$, the subsystem has three input ports and on output port. Inside the Force subsystem there is defined the block *Add*, that is a *Sum* block type. Clearly, other blocks can be defined inside this subsystem,

```
<ModelInformation Version="0.9">
  <Model Name="my_Diagram">
    ...
      <System>
        ...
      < \System>
    ...
  < \Model>
< \ModelInformation>
```

Fig. 15. Starting the System Diagram Specification.

```
<ModelInformation Version="0.9">
  <Model Name="my_Diagram">
    ...
      <System>
        <Block Name="ControlMode" BlockType="Switch">
          <P Name="Position"> [545, 557, 620, 653] </P>
          <P Name="Criteria"> u2 ≥ Threshold </P>
          <P Name="Threshold"> 10 </P>
          ...
        < \Block>
      < \System>
    ...
  < \Model>
< \ModelInformation>
```

Fig. 16. (Basic) Block Diagram Specification.

whose definition ends with the tag < \System>. After this tag, the description of the parent system continue.

All these information are processed from the parser in order to obtain the corresponding SpaceEx model, as explained in Section 5.

4 SpaceEx Verification Tool

SpaceEx is a verication platform for hybrid systems. The goal is to verify (ensur beyond reasonable doubt) that a given mathematical model of a hybrid system satises the desired safety properties. The basic functionality is to compute the sets of reachable states of the system. SpaceEx consists of three components, see Figure 18:

1. The *Model Editor* is a graphical editor for creating models of complex hybrid systems out of nested components. It produces model les in SpaceExs `sx` format.
2. The *Analysis Core* is a command line program that takes a model file in `sx` format, a configuration file that specifies the initial states, the scenario

```
<ModelInformation Version="0.9">
  <Model Name="my_Diagram">
    ...
      <System>
        ...
        <Block Name="Force" BlockType="SubSystem">
          <P Name="Ports"> [3, 1] </P>
          <P Name="Position"> [545, 557, 620, 653] </P>
          ...
          <System>
            <Block Name="Add" BlockType="Sum">
              ...
            < \Block>
            ...
          < \System>
          ...
        < \Block>
      < \System>
    ...
  < \Model>
< \ModelInformation>
```

Fig. 17. (SubSystem) Block Diagram Specification.

(linear verification, simulation, etc) and other options, analyzes the system and produces the output file.

3. The *Web Interface* is a graphical user interface with which one can comfortably specify initial states and the other analysis parameters, run the analysis core, and visualize the output graphically. The web interface is browser based, and accesses the analysis core via a web server. which may running remotely or locally on a virtual machine.

Strictly speaking, the SpaceEx analysis core is not a single tool but a development platform on which several different verication algorithms are implemented. Each algorithm may come with its own set representation, apply to its own class of models, and produce a different kind of output, so we refer to such a bundle as a scenario. Currently, two scenarios are implemented:

- The PHAVer scenario implements the basic algorithm from the tool PHAVer. It applies to Linear Hybrid Automata, which are hybrid systems with piecewise constant bounds on the derivatives.
- The LGG Support Function scenario implements a variant of the Le Guernic-Girard (LGG) algorithm. It applies to hybrid systems with piecewise affine dynamics with nondeterministic inputs.

4.1 Modeling Hybrid Systems in SpaceEx.

SpaceEx models are stored in the `sx` format, an XML based format for which there is a graphical model editor. Please refer to the examples provided on the

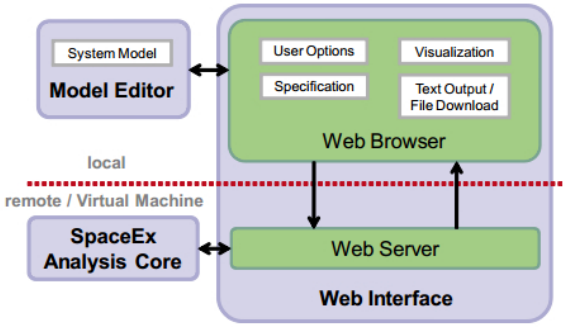


Fig. 18. Software architecture of the SpaceEx platform.

download page and on the virtual machine. `sx` models are similar to the hybrid automata known in literature, except that they provide a richer mechanism of hierarchy, templates and instantiations. An `sx` model consists of one or more components. When SpaceEx reads an `sx` le, it translates the components into either an automaton or into a network of automata in parallel composition.

4.2 Components.

A model is made up of one or several *components*. There are two types of components: A *base component* corresponds to a single hybrid automaton, and is dened by its locations and transitions. A *network component* consists of one or more instantiations of other components (base or network) and corresponds to a set of hybrid automata in parallel composition. Every component has a set of *formal parameters*. A formal parameter may be:

- a continuous variable with arbitrary dynamics,
- a continuous variable with constant dynamics, i.e., it does not change its value over time. It may be assigned a value, like $3,1415$, when it is instantiated in a network component. Then it becomes what is commonly referred to as a "constant",
- a synchronization label. Every transition is associated with a label.

A formal parameter is part of the *interface* of a component, unless it is declared as *local* to the component.

Note: All symbols (variables, constants or labels) that are used in describing locations or transitions must be declared as formal parameters of the corresponding component. Symbols that are not of interest outside of the component can be declared as local.

Example 5. The interface of a bouncing ball model is shown in Figure 19. The continuous state variables x and v are declared as variables with arbitrary

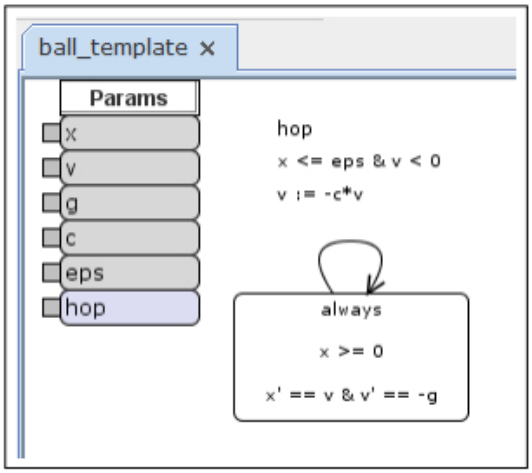


Fig. 19. A bouncing ball model with its formal parameters: state variables x and v, constants g, c, and eps, and label hop.

("any") dynamics. The constants g, c, and eps are declared as variables with constant dynamics. They will be assigned values when the template is instantiated inside a network component. The last parameter is the synchronization label hop.

4.3 Base Components and Semantics.

A base component in the model is translated by SpaceEx into a *hybrid automaton*. It consists of a graph in which each vertex, called *location*, is associated with a *ow*, which is a set of differential equations (or inclusions) that denes the time-driven evolution of the continuous variables. A *state* consists of a location and a value for each of the continuous variables. The edges of the graph, called *transitions*, allow the system to jump between locations, thus changing the dynamics, and instantaneously modify the values of continuous variables according to an *assignment*. The jumps may only take place when the values of the variables satisfy the constraints of the *guard* of the transition. The system may only remain in a location as long it satises the constraints of the *invariant* associated with the location. All behavior originates from a given set of *initial states*. An *execution* of the automaton is a sequence of discrete jumps and pieces of continuous trajectories according to its dynamics, and originates in one of the initial states. A state is *reachable* if an execution leads to it. Given a set of forbidden states, the system is *safe* if the bad states are not reachable.

Dynamics and Constraints. The ow of a location is a set of differential equations. It depends on the scenario which types of constraints are allowed. The LGG scenario accepts afne dynamics of the form

$$(d/dt)x = Ax + Bu + b_0,$$

where u is a set of nondeterministic inputs. Constraints on u can be included in the invariant of the location. The `sx` format does not currently support vector/matrix notaion. A system of the above form is described by the expression

$$\begin{gathered} x1' == a11 * x1 + ... + a1n * xn + b11 * u1 + ... + b1m * um \ And \\ x2 == a21 * x1 + ... + a2n * xn + b21 * u1 + ... + b2m * um \ And \\ \cdots \\ xn == an1 * x1 + ... + ann * xn + bn1 * u1 + ... + bnm * um \end{gathered}$$

where the prime behind a variable denotes its derivative. Similarly, a transition may modify the variables with an assignment of the form

$$x := Ax + Bu + b_0;$$

where u is a set of nondeterministic inputs that is constrained only by the invariant. The assignment can be described in two forms: either with primes

$$\begin{gathered} x1' == a11 * x1 + ... + a1n * xn + b11 * u1 + ... + b1m * um \ And \\ x2 == a21 * x1 + ... + a2n * xn + b21 * u1 + ... + b2m * um \ And \\ \cdots \\ xn == an1 * x1 + ... + ann * xn + bn1 * u1 + ... + bnm * um \end{gathered}$$

where the prime denotes the value after the transition, or by the equivalent expression

$$\begin{gathered} x1 == a11 * x1 + ... + a1n * xn + b11 * u1 + ... + b1m * um \ And \\ x2 == a21 * x1 + ... + a2n * xn + b21 * u1 + ... + b2m * um \ And \\ \cdots \\ xn == an1 * x1 + ... + ann * xn + bn1 * u1 + ... + bnm * um \end{gathered}$$

Variables that are not assigned explicitly are supposed to remain constant during the transition. Guards and invariants can be arbitrary convex linear constraints on the variables, where conjunctions are denoted by an ampersand (&). For example:

$$a * x + b * y \ == \ 1 \ \& \ c \ <= \ z \ <= \ d$$

A universal constraint can be denoted with `true`, and an unsatisable constraint by `false`.

4.4 Network Components.

A network component allows one to instantiate one or more components (base or other network components), connect them via their variables and labels, and assign values to their constants.

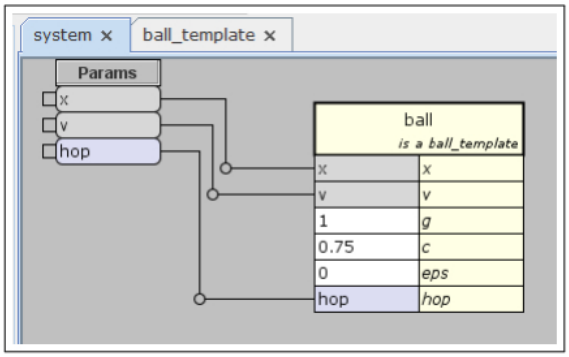


Fig. 20. The bouncing ball template instantiated in the network component system. The constants are bound to the desired values, the other formal parameters are passed on to the interface of system.

Instantiating Components. When a component A is instantiated inside a network component B, a copy of A is created and added to the contents of network B. The copy must have a name that is unique within B, say A1. It is a copy by reference, so that any later modications to A will also apply to A1. Component A can be instantiated several times inside B, say as A1, A2, A3, etc. When the sx model is parsed by SpaceEx, each base component is translated into a corresponding hybrid automaton, and each network component is translated into the parallel composition of its subcomponents. Semantically, the parallel composition of hybrid automata is itself a hybrid automaton. In SpaceEx, this composition is carried out on the y, so that only the reachable parts of that automaton are actually created in memory. When instantiating a component A in network B, we must specify what happens to each of the formal parameters in its interface. This is called a *bind*. Every formal parameter of A must be bound to either a formal parameter of B or to a numeric value.

Example 6. Figure 20 shows the network component system, which is made up of an instantiation of the base component ball template. The instantiation is called ball. The formal parameters x, v, and *hop* are bound to formal parameters x, v, and *hop* of system. The constants g, c, and *eps* are bound to numeric values.

Connecting Components. Components inside a network can be connected by binding their variables or labels to same symbols in B. Whether this connection is in parallel or in series is not fixed by the interface. It depends entirely on the restrictions the components make on the variables.

Example 7. Figure 21 shows a simple rst-order low-pass lter. It has u as input and x as output, c is a value determining the cut-off frequency. The network component shown in the right side of the figure consists in two instantiations of this filter put in series. The network has input u_{in}, the output x_{out}, and the

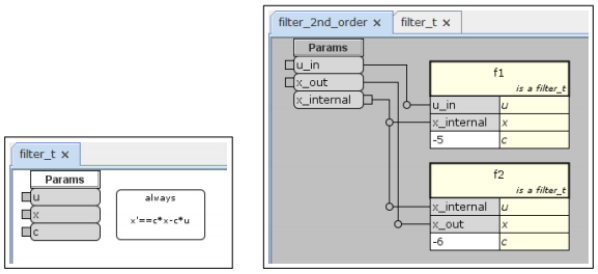


Fig. 21. A second order lter constructed using two instantiations of a rst order lter.

internal variable $x_{internal}$. The filter are put in series by binding the output of the first filter and the input of the second filter to the same variable $x_{internal}$. When this model is read in SpaceEx, it creates a hybrid automaton with three variables: u_{in}, x_{out}, and $x_{internal}$. Note that $x_{internal}$ was chosen to be local to the second order filter. This means that it does not gure in its interface, and should the second order filter be used in a network together with other components, they will not have access to that variable.

Note: Binding two or more variables to the same variable in a network component means that they became literally the same variable. Their dynamics can ben defined in one or several of the component as long as this creates no contraddictions.

4.5 Causality and Nondeterminism.

SpaceEx models have nondeterministic, acausal semantics: any component can declare any variable as one its formal parameters, and impose constraints (including equalities) on the variable and its derivative. Different components can impose constraints on the same variable, at the same time or alternation. In certain application areas, like mechanics or electric circuits, this allows one to decompose models into reusable building blocks, or build models directly from first principles. For example, one component could impose $\dot{x} \leq 0$ and another $\dot{x} \geq 0$. The resulting behaviour has to satisfy both, so $\dot{x} = 0$. If the constraints contradict each other, resulting in, e.g., $\dot{x} \in \emptyset$, there is no solution to the differential equations (or inclusions) and thus there might not be any trajectories after a certain point in time. We say that "time stops" in the model. This may or may not be a modeling error. Usually, there should be at least one trajectory over infinite time for every initial state. But in a nondeterministic system, this doesn't mean that all trajectories from the initial states range over innite time. Especially when overapproximations are used to simplify a model, states in which time stops may be reachable. As long as this is an artifact of the overapproximation, this poses no problem.

Note: Modeling errors may "stop time" when contradicting constraints render differential inclusions or transition assignments unsatisable. The model may

then be trivially safe as an artifact of that modeling error. Care should be taken to ensure time passes as intended by the modeler.

It is recomended that the modeling process is accompanied by steps that ensure that time indeed passes up to the desidered point. While a formal verification of this liveness property is hard and beyond the capability of SpaceEx, the following informal procedure can help to detect at least crude modeling errors:

1. Pick a time point T that is large relative to the time constants in the system.
2. Create a monitor automaton with a variable d, flow $\dot{d} = 1$ and invariant $d \leq T$.
3. Put the system in parallel with the monitor and verify that from the initial state $d = 0$ the state $d = T$ is reachable.

4.6 Inputs, Outputs and Controlled Variables.

Because of the acausal semantics, there are no inputs and outputs in SpaceEx models per se. For compositional reasoning, there is the notion of a *controlled variable*, which is closely related, and SpaceEx enforces compositional semantics for this purpose.

Note: Users who do not care about compositional reasoning can simply declare all variables as controlled in all components.

We now give a brief description of compositional reasoning and how it employs controlled variables. The following notions about compositional reasoning apply to continuous variables as well as to events (synchronization labels), but for brevity we limit the discussion to the continuous part. Recall that a component H with a variable x restricts the behavior of x over time. This restriction is expressed in the constraints that invariants, flows, guards and assignments of H pose on x and $\dot{x}$. A compositional framework guarantees the following: the behavior of x in a network component that contains H is a subset of the behavior of x in H by itself. So if the behavior of x in H is safe, then it is safe also in any other component that includes H. To make the above guarantee, one needs to forbid that other components modify x beyond what is possible in H itself. Some frameworks achieve this by declaring x controlled in components that have as a subcomponent H and uncontrolled everywhere else. In addition, the following semantic restrictions are imposed on all components G in which x is uncontrolled:

- When G takes a transition that does not synchronize with a transition of H, it must not change the value of x in that transition. To ensure this SpaceEx adds the constraint $x' == x$ to such transitions in G.
- When G remains in a location and time passes, x must be able to take on any value that satisfies the invariant of G. To ensure this SpaceEx adds self-loops to G that assign x nondeterministically to the allowed values.

Note that no restrictions are imposed on components in which x is controlled. For compositional reasoning, there can be at most one base component that controls a variable, and it must be uncontrolled in all other base components. For

network components, the model editor deduces automatically whether a variable is controlled or not. In the semantics that result from the above restrictions, uncontrolled variables can be loosely understood as *inputs*, and controlled variables as *outputs*.

Note: Users who wish to apply compositional reasoning need to declare every variable as controlled in at most one base component, and as uncontrolled in all other base components. SpaceEx enforces compositional semantics by adding the appropriate constraints and self-loop transitions for every variable that is declared as uncontrolled.

4.7 Initial and Forbidden States.

The specification of a reachability problem includes the set of initial states, from which all behavior of the system originates. A set of states can be specified in SpaceEx as a boolean combination of *location constraints* and linear constraints over the variables (see invariants and guards). Conjunctions are denoted by an ampersand (&) and disjunctions by a vertical bar (|). A location constraint is of the form

$$loc(\ <\ \textit{base component name}\ >\ )\ ==\ <\ \textit{location name}\ >;$$

$$loc(\ <\ \textit{base component name}\ >\ )\ !=\ <\ \textit{location name}\ >;$$

where the first form denotes a single location in the given component, and the second form denotes all other locations of that component.

Dot notation and context dependent lookup. Component and variable identifiers are instantiated in SpaceEx such that they are unique for each instantiation of a base component. This is achieved by prepending their names with the names of the network components that contain the base component, separated by a period (.). To avoid unnecessarily long names, SpaceEx uses context dependent lookup with respect to the component that is selected for analysis. Component names do not need to be prepended if the identifier is unique within that component.

Example 8. Suppose a base component A has a local variable x and locations a and b. A network component B consists of two instantiations of A called $A1$ and $A2$. To specify that $A1$ is in location a with $x = 1$ and $A2$ is in location b with $x = 2$, one can use the constraints

$$loc(B.A1) == a\ \&\ B.A1.x == 1\ \&\ loc(B.A2) == b\ \&\ B.A2.x == 2.$$

Suppose C is a network component containing an instantiation $A3$ of base component A, and D is a network component containing and instantiation $C1$

<?xml version=“1.0” encoding=“ISO-8859-1”?>.

Fig. 22. XML Header.

<sspaceex xmlns=”http://www-verimag.imag.fr/xml-namespaces/sspaceex” version=”0.2” math=”SpaceEx”>
...
< \sspaceex>

Fig. 23. Starting Model Specifications.

of C. We can define initial states for $A3$ with $loc(D.C.A3) == a$ & $D.C.x == 1$. If D is the component to be analyzed, we can omit the prefix D: and write $loc(C.A3) == a$ & $C.x == 1$. If $A3$ is the only component inside C that has a variable called x, we can omit the prefix for the variable and write $loc(C.A3) == a$ & $x == 1$. If D contains only one component with the name $A3$, we can omit its prefix as well, which leaves us with $loc(A3) == a$ & $x == 1$.

If the component to be analyzed is a base component, there is no need to specify the component name, and one can just write $loc() == a$ instead.

4.8 The Representation of a SpaceEX model in XML File Format.

At the beginning of this section we referred to the SpaceEx file as `sx` format. Actually, this format is written as an XML file. In the following we will show how a SpaceEx model is stored by .xml file.

The first row, as usual, is used to define the xml version and the encoding, as Figure 22 shows.

The effective description of the model starts with the tag *spaceex*, as shown in Figure 23

Now the model must be defined by specifying the basic and the network components that forms the system. The specification of a basic component is shown in Figure 24, where an hybrid automaton consisting in one location, two variables ($In1$ and $Out1$) and a constant ($Gain$) is defined. The specification also describes the invariant associated to the location ($Out1 = In1 * Gain$) and some graphical informations, as the position of the variables and their placement. Notice that, the described automaton models a Simulink Gain Block Type that Figure 25 shows the corresponding graphical representation (as the automaton appears on the SpaceEx Editor).

In Figure 26 is depicted the decleration of a Network Component. In particular, a Network Component called “MyModel” is defined with three variables (x and $AddOut1$ uncontrolled, y controlled), a constant $Gain$ and a *bind* “Gain1” to the basic component *Gain*. There is the following mapping among the variables, defined by the tags *map*

- The $In1$ variable belonging to the basic component $Gain$ is mapped to the parameter $AddOut1$

```
<sspaceex xmlns="http://www-verimag.imag.fr/xml-namespaces/sspaceex" version="0.2" math="SpaceEx">
    ...
    <component id="Gain">
        <param dynamics="any" d2="1" d1="1" local="false" type="real"
        name="In1" controlled="false" placement="west" x="440" y="325" />
        <param dynamics="any" d2="1" d1="1" local="false" type="real"
        name="Out1" controlled="true" placement="east" x="440" y="325" />
        <param name="Gain" type="real" local="false" d1="1" d2="1"
        dynamics="const" />
        <location id="1" name="loc" x="210.0" y="110.0"¿
            <invariant>Out1=In1*Gain</invariant>
        </location>
    </component>
    ...
< \sspaceex>
```

Fig. 24. Base Component Definition.

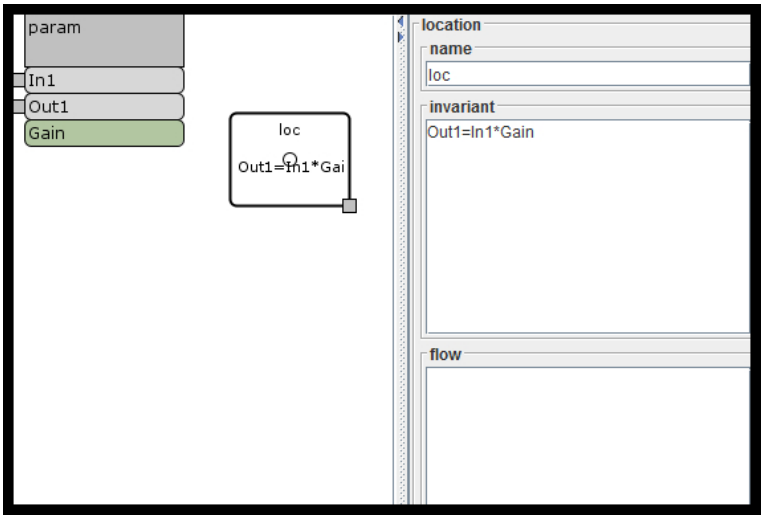


Fig. 25. How the Automaton specified in Figure 24 appears on the SpaceEx Editor.

- The $Out1$ $Gain$ variable is mapped to the parameter y
- The constant parameter $Gain$ is mapped to the value 4

Notice also the presence of the tag *link* inside the mapping: the values inside this tag define the graphical shape of the line that connect two mapped variables.

Figure 27 shown the entire network component, whose part of the definition is depicted in Figure 26

```
<sspaceex xmlns="http://www-verimag.imag.fr/xml-namespaces/sspaceex" version="0.2" math="SpaceEx">
    ...
    <component xmlns="http://www-verimag.imag.fr/xml-namespaces/sspaceex"
    id="MyModel">
        <param dynamics="any" d2="1" d1="1" local="false" type="real"
        name="x" controlled="false" placement="west" x="135" y="248" />
        <param dynamics="any" d2="1" d1="1" local="false" type="real"
        name="y" controlled="true" placement="east" x="631.0" y="313.0" />
        <param dynamics="any" d2="1" d1="1" local="false" type="real"
        name="AddOut1" controlled="false" placement="east" x="320" y="207"/>
        <param name="Gain" type="real" local="false" d1="1" d2="1"
        dynamics="const" controlled="true" /¿
        <bind component="Gain" as="Gain1" x="300.0" y="207.0"
        width="20" height="111">
            <map key="In1"> x
                <link>192.0,228.0,192.0,248.0</link>
            </map>
            <map key="Out1">GainOut1</map>
            <map key="Gain">3</map>
        < \bind>
        ...
< \sspaceex>
```

Fig. 26. Network Component Definition.

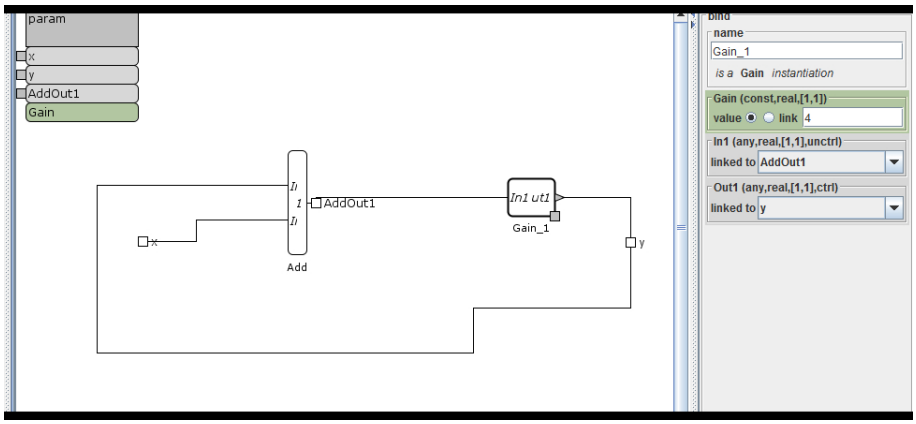


Fig. 27. How the Network partially specified in Figure 26 appears on the SpaceEx Editor.

5 Simulink to SpaceEX SemiAutomatic Translator (SL2SX).

This section describe how the tool *SimuLink To SpaceEX Automatic Translator (SL2SX)* is implemented. The core of the process is a parser that read the

Simulink Diagram written in XML format (see Section 3) and write a corresponding (equivalent) SpaceEx Model also written in its proper XML format (see Section 4).

The tool is designed in order to achieve two goals:

1. To obtain a composition of Hybrid Automata, by the means of Basic and Network component (see Section 4), that is equivalent to the Simulink diagram in input. When we speak about equivalence, we mean Reachaibiliy Equivalence, i.e. starting from a set of initial states, each state reachable in a Simulink diagram SL is also reachable in the corresponding SpaceEx model SX, and viceversa.
2. The second goal can appear secondary, but it is not: the tool is tought to keep the graphical correspondence with the input simulink diagram. This aspect is very useful, because it is very common that the design of a real system is very complicate, with a lot of blocks, connections and variables. In order to obtain a corresponding SpaceEx model that a real user can really maneage, a fundamental aspect is that all the corresponding blocks are in the same position as the original one and also for the connections and the variables name. This help the user to made real experiment and comparisons between the results coming out from the the tools.

The basic idea behind the tool, in order to achive these goals, is to express each Simulink block (not SubSystem type) as a Basic Component, while each SubSystem block type is expressed as a Network Component. The main System became the main Network Component.

Clearly the tool cannot translate all the Simulink blocks: some will be translated in the future (the implementation is written in a way that allows to write extensions easily), while others maybe cannot be translated at all. When the translator meet a block whereby is not defined a corresponding Basic Component, the tool does not ignore it at all: all the basic informations (as number of input and output variables, connections with other blocks, positions, dimension, etc) are retrived. The corresponding basic component, is an hybrid automaton whith one location (without invariant and flow) and a number of parameters corresponding to the number of the block variables. In this way, the resulting model will preserve the schema of the input Simulink diagram. On the editor, the user can check if the particular component really implement the corresponding block, by a mouse click: if the block is not supported, the editor shows a note (*"This block is not supported by the current version"*).

5.1 Diagram Class.

The source code is written by Java Language and uses the SAX parser. The choice of the language and the parser come from the fact that all the applications around the SpaceEx platform, like the editor, the filebuilder, the graphical interface, whose computational time is not a central point, are written in Java. More, by using Java make possible to use some routine alreaday written and included in the platform, and this allows to mantein and integrate our application

in the SpaceEx platform in a natural way, also avoiding to rewrite part already implemented.

We now start to give the description of how our application is implemented. Figure 28, shows the design (in terms of the Classes and their attributes) of the application. The next paragraphs explain in detail the rules of each class and how the simulink diagram information are stored in the various attributes.

Variable Class. Recalling that each Simulink block is associated to a list of input and output variables, this class is used to model each of these variables. Once created an object corresponding to a variable, the other attributes of the class are filled by using the information about blocks connections (retrivable under the part of the XML file identified by the *Line* tag, as explained in Section 3). In particular, the class stores the name of the variable connected to the modeled variable (used to define the *Map* in the `SX` model) and also information about the graphical shape of the line that will appears in the SpaceEx Editor.

- **Attributes:**
  - *name*. Name of the modeled variable. The convention is that, when the block is not an InPort or OutPort type, the name of the output variable is given by postfix the string "Out" following by the number of the variable to the name of the block. For example, for a Sum type block called "Add", the corresponding output variable is called "AddOut1". While in the case of an inport or an outport, the name is the same of the block name. Notice that each of these variable will be used in order to define the SpaceEx *parameters*.
  - *varMap*. Name of the variable connected to the modeled variable. Used to define the *map* in the Network Component of the SpaceEx model where the variable exists.
  - *points*. This attributes is used in order to store the value belonging to the tag *Points* in the file that described the Simulink diagram. This value is used to define the shape of the graphical line that connect the blocks, and will be processed in order to obtain the same shape in the SpaceEx editor.
  - *link*. Contains the shape of the SpaceEx editor graphical line, obtained from the value of the attribute *point*.
  - *xPos*. Contains the x-coordinate of the graphical position of the variable.
  - *yPos*. Contains the x-coordinate of the graphical position of the variable.
  - *controlled*. Used to distinguish if the corresponding parameter will be defined as controlled or not.
  - *input*. If true, means that the modeled variable is a block input variable.
  - *output*. If true, means that the modeled variable is a block output variable.
  - *inPort*. If true, means that the modeled variable corresponds to an InPort block type.
  - *outPort*. If true, means that the modeled variable corresponds to an OutPort block type.

  - *placement*. Identify the graphically side where the parameter will be placed (west or east) in the SpaceEx model.
- **Responsabilities:**
  1. Model a variable corresponding to an input/output variable of a block, or corresponding to an In/OutPort block type.
  2. Keep track to che connection between the modeled variable and another variable.
  3. Store graphical information (parameter position and line connection shape).

SLBlock Class. This Class is used to model Simulink Block. This mean that for each block that defines a Simulink diagram, there will be the instantiation of this class whose part of the attributes are clearly the same that defines a block (see Section 3). Moreover, the class has attributes that says if the modeled block is supported by the `SL2SX` current version and if it inherits default parameters. About block default parameters (see Section 3), this class is also used in order to store this information: this means that for each definition of a standard block, an instantiation of this class is created in order to store the default characteristic values of the block.

- **Attributes:**
  - *name*. Name of the modeled block. It is retrived from the value of the attibute *Name* associated to the tag *Block* in the Simulink file.
  - *type*. Type of the modeled block. It is retrived from the value of the attibute *BlockType* associated to the tag *Block* in the Simulink file (for a list of Simulink Block Type see Section 3).
  - *ports*. It is retrived by reading the value associate to the tag *P Name="Ports"*, and it is later used in order to estabilish the number of the input and output variables of the block, then stored in the attributes *inPortsNum* and *outPortsNum*, resp.
  - *position*. Retrived by reading the value associate to the tag *P Name="Position"*, and it is later used in order to estabilish the dimension and the position of the block (information then stored in the attributes *x1Pos*, *y1Pos*, *widht* and *height*).
  - *value*, *gain*, *inputs*, *outputs*, *criteria*, *threshold*, *externalReset*, *lowerValue*, *upperValue*, *gotoTag*: attributes used to store the characteristic parameters of the different block type (see Section 3).
  - *defaultB*: if true, the modeled block inherits the default parameters defined for the block type.
  - *supported*: if true, the modeled block is supported by the current version of the tool: used during the writing of the SpaceEx model to decide wether or not add a SpaceEx note that advises that the modeled block is not supported.
- **Responsabilities:**
  1. Model a block inside the main system or in a subsystem of the Simulink diagram.

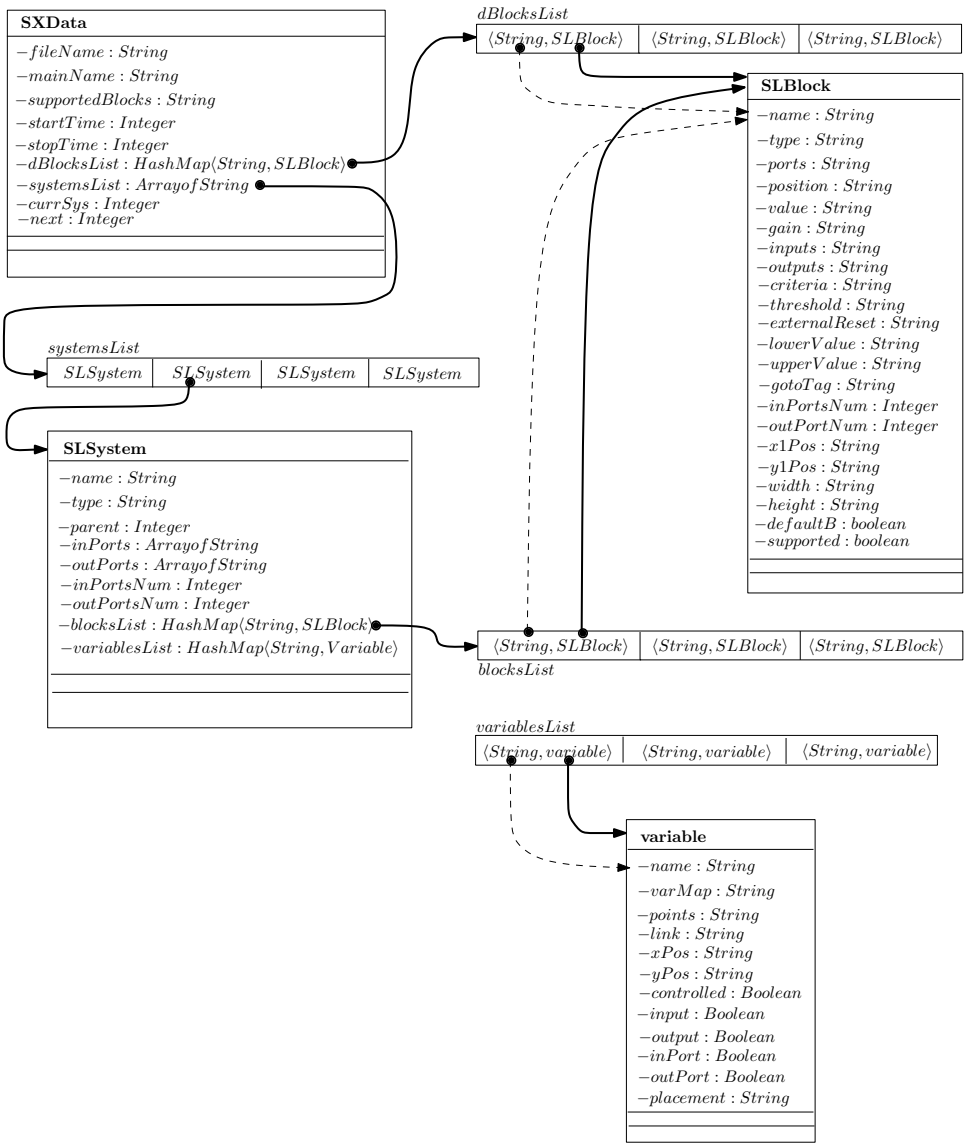


Fig. 28. Classes and Relations.

2. Model the definition of default parameters for a given block type.
3. Store information about the characteristic parameters of the modeled block.
4. Store graphical information (block position and dimension).

SLSystem Class. This Class is used to model the main System of the Simulink diagram or block whose block type is "SubSystem". This mean that for the

main system and for each subsystem block, there will be the instantiation of this class. Each class instantiation contains information about the parent System (the subsystem that contains the modeled subsystem block), about the blocks present in the subsystems and about its variables (parameters).

- **Attributes:**
  - *name*. Name of the modeled (sub)system. It is retrived from the value of the attibute *Name* associated to the tag *Block* in the Simulink file, when the block type is *SubSystem*.
  - *type*. Just to distinguish the main system to all the other subsystems.
  - *parent*. Together with the attributes *systemsList* defined in the Class *SXData*, allows to get the subsystem that contains the modeled subsystem.
  - *inPort*. Used to store the name of the InPort conteined in the modeled subsystem, used in order to estabilish the parameters of the corresponding Network Component in the SpaceEx model.
  - *outPort*. Similar to the former, but for OutPort.
  - *blocksList*. Is the list of all the blocks that forms the modeled subsystem. It is implemented by an HashMap, in order to get the corresponding block by the name (i.e. the name is the key of the structure). This list is used in order to define the corresponding Basic Component in the SpaceEx model.
  - *variablesList*. Is the list of all the variables in the subsystems. Also this list is implemented by an HashMap, and the variable can be accesses by name. The variables list is used in order to define the SpaceEx parameters of a Network Component, whose features (dynamics, type, position) is retrived from the corresponding variables modeled by the class variable.
- **Responsabilities:**
  1. Model the main system and a SubSystem block type of the Simulink diagram.
  2. Store information about the design of the subsystem, like blocks, variables, InPorts and OutPorts.

SXData Class. This Class is the component on the top of the implementation. All the higher-level informations about a Simulink diagram are managed by this class that, by an abstract point of view, describes the whole system as a tree, where the main system is the root of the tree, each node not leaf is a subsystem and each leaf is a block. The attributes of this class are used to store information about the name of the file where the Simulink diagram is written, simulation parameters (start and stop simulation time), a list that contains the default block parameters and the list of the systems in the diagram, that in fact implements the former mentioned logical tree.

- **Attributes:**
  - $fileName$. $String$. Contains the name of the file where the Simulink diagram is phisically stored.

- *supportedBlocks*. *String*. Contains the block type name supported by the current version of *SL2SX*, separated by a white space. When the translation of a new blocktype is addes, the programer has to add the blocktype to this attribute.
- *mainName*. *String*. Contains the name of the Simulink diagram main system.
- *startTime*. *String*. Contains the value of the starting time simulation.
- *stopTime*. *String*. Contains the value of the stop time simulation.
- *systemsList*. *Array of String*. Contains the name of the mainsystem (first position) and the subsystems of the Simulink diagram. This list is then used in order to write the corresponding Network Components. Notice that, because the systems are stored from the first position of the list to ahead, in order to be able to define in the right way the corresponding SpaceEx model, the list has to be scanned from the last element: in this way we are sure that all the bind of a Network Component are yet defined.
- *dBlockList*. *HashMap*. For each definition of a default block parameters (see Section 3), there is a corresponding element in this structure (accessible via the name of the block type) where the parameters are stored.
- *currSys*. *Integer*. Used to keep track of the system currently analyzed by the parser (i.e. the index of *systemsList* where the system is stored).
- *next*. *Integer*. Used to keep track of the next available position in the *systemsList* array. Notice that this is necessary because of the hierachy: the current system can be a father of a system already analyzed and stored one position after it ($currSys + 1$), so the next available position is not just one position ahead in the list, but two position after the current ($next = currSys + 2$).

– **Responabilities:**
  1. Model the the Simulink diagram.
  2. Store simulation parameters.
  3. Store block parameters default.
  4. Contains the list of the systems that are involved in the diagram.

6 Cases Study

In this section we give concrete examples of use of the tool SL2SX that cover all the process phases, till the verification by SpaceEx of the obtained hybrid automaton and the comparision between the simulations. To be more precise, we will present three examples: the first one to show a case where SL2SX is able to fully translate the model (user intervention is not required). The second and third examples depict semi-automatic cases, where the user must adapt a bit the SpaceEx models to obtain the right results. Notice that, the last cases comes directly from the official examples library of Simulink (`http://www.mathworks.com/help/simulink/examples.html`).

6.1 DCMotor: Full Automatic Example

Figure 29 depict the Simulink diagram of the example, stored in the file "DC-Motor.mdl".

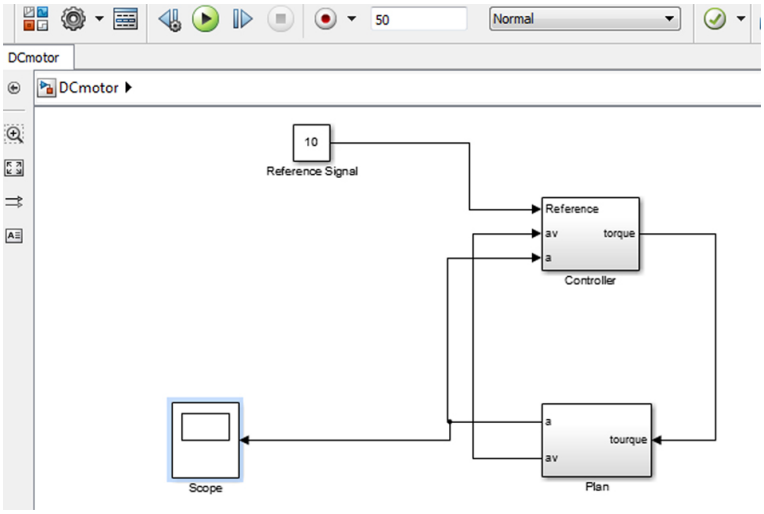


Fig. 29. A Simulink Diagram

The model consists in four blocks:

1. *Reference Signal*: is a costant block type, whose associated value is 10.
2. *Controller*: is a subsystem block type, with three input variables ($Reference$, av and a) and the single output $torque$.
3. *Plant*: is another subsystem block type, with the variable $torque$ as input and the two output a and av.
4. *Scope*: a scope block type, used to monitor the values in the time of a variable.

The blocks are connected in this way:

- The constant value (10) is one of the input of the *Controller* subsystem, via the variable $Reference$.
- The controller input variables a and av are connected to the plant output variables a and av.
- The plant input $torque$ is connected to the controller output variable $torque$.
- THe input of the scope is the variable a: this means that, by this block, it is possible to monitor the evolution in the time of the variable a, that is the output of the plant and used as feedback by the controller.

Subsystem block types allow the designer to model hierarchical system. Indeed, by looking inside the two subsystem blocks of the example (i.e. *Controller*

and *Plant*), it is possible to identify another Simulink diagram, with the proper blocks and connections. For example, Figures 30 and 31 show the diagrams inside the controller and the plant respectivly.

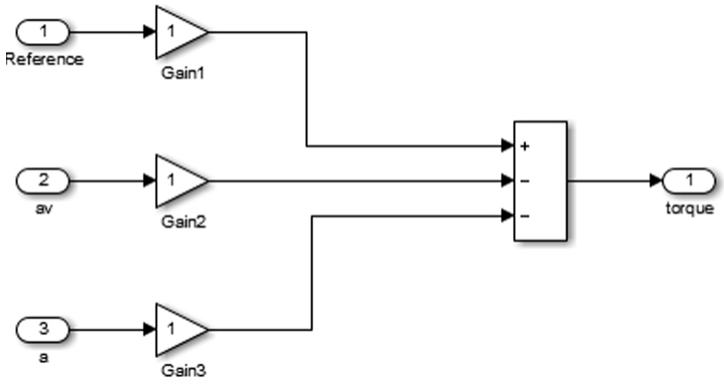


Fig. 30. Inside *Controller* SubSys

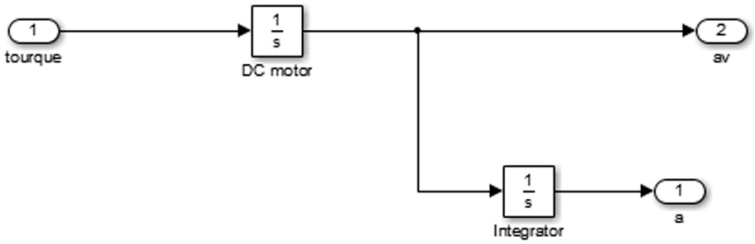


Fig. 31. Inside *Plant* SubSys

The controller subsystem consists in three gain blocktype, whose outputs are the inputs of a sum blocktype that subtracts second and third input from the first one, while the plant consists in two integrator, to compute first and second derivative of the input variable *torque*.

Translation by SL2SX The Simulink diagram of the example above is stored in the file *DCMotor.mdl*, hence we first need to export it to .xml format. MATLAB allow the users to do this by the simple *save_system* command, as described in Figure 32.

```
>> save_system('DCMotor', 'DCMotor.xml', 'ExportToXML', true)
```

Fig. 32. MATLAB command to export in .xml

As effect of this command, we obtain the file *DCMotor.xml*, that is ready to be treated by SL2SX.

Figure 33 shows what appears once our application is launched.

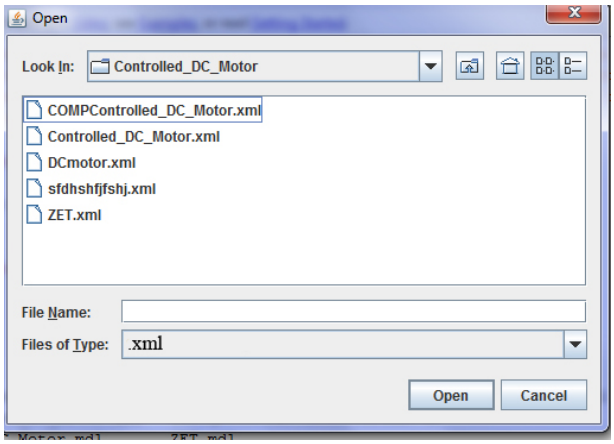


Fig. 33. Dialog Box to choose the .xml file of the Simulink Model

In particular is a dialog box that ask the user to choose the .xml file where the Simulink diagram to convert is stored. In our case, we select the file *DCMotor.xml* (see Figure 34).

Once the file is selected, the translation starts and the tool provides status information to the user, as depicted in Figure 35, where the the last message says that the process was correctly done, and hence the file *SX_DCmotor.xml* (the SpaceEx Model) is now available for the user.

Syantax Comparision As we sayed before, our goal is not only to obtain an hybrid automaton that is semantically equivalent to the Simulink diagram in input, but we want to obtain also a kind of *syntactically-equivalence* between the two models. In order to achive this, we want that the translation preserves

1. Variables and their names (*Reference, av, a, torque, ...*)
2. Graphical Shape (dimension and position of the blocks, line-connections, ...)
3. Hierarchy (Controller and Plan have other components inside, ...)
4. The simulation time (50)

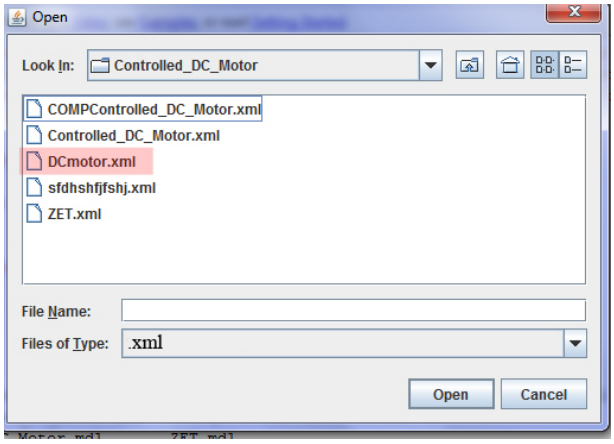
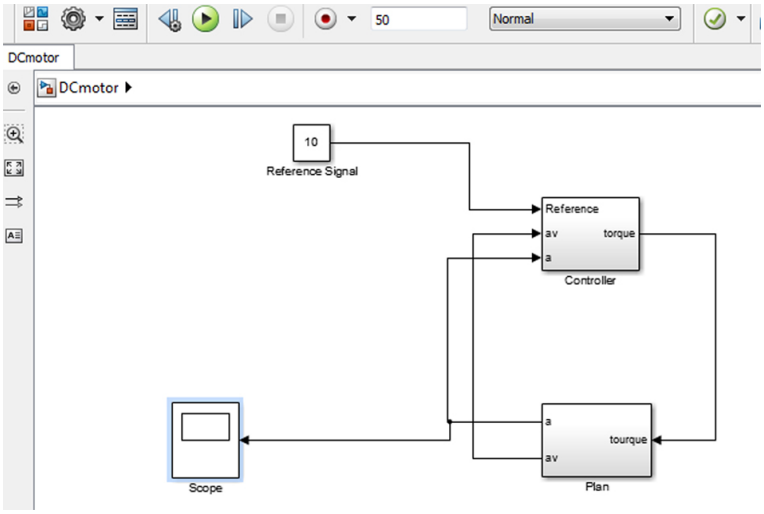


Fig. 34. Open DCMotor.xml

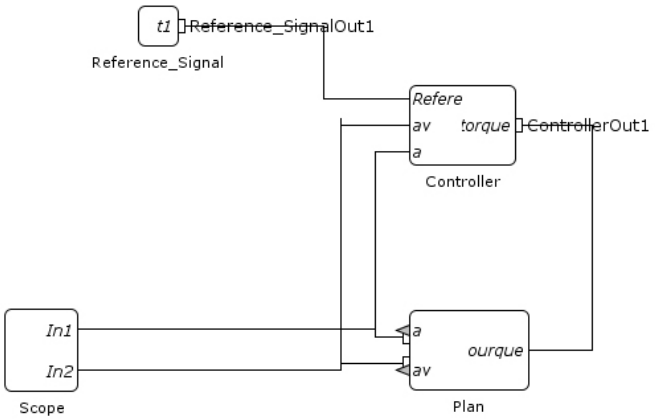
```
>> STEP 1. Choose the Simulink Diagram File in .xml Format...

         ------------------------------------------------------------
           The current version supports the following Blocks Type:
           System SubSystem Inport Outport Constant Gain Sum Product Integrator Switch From Goto DeadZone Trigonometry
         ------------------------------------------------------------

>> STEP 2. Starting translation from SimuLink to SpaceEx format . . .
  >> 2.1. Begin.
               From SL xml --> Internal Data Structure.

  >> 2.1. Done.

  >> 2.2. Begin.
              Postprocessing Internal Data Structure.
  >> 2.2. Done.

>> STEP 3.   Write the corresponding SpaceEx Model on file.
  >> 3.1 Begin.
             From the Internal Data Structure --> .xml SpaceEx Model.
  >> 3.1 Done.
  >> 3.2 Begin.
             From the Internal Data Structure --> .xml SpaceEx Model Configuration.
  >> 3.2 Done.
```

Fig. 35. Messages from the tool during the translation phase.

To check if our requirements are met, we open the SpaceEx model *SX_DCmotor.xml*, by the SpaceEx Editor. Figure 36 shows the comparision between the main Simulink diagram and the top level SpaceEx component.



(a) Simulink Main System



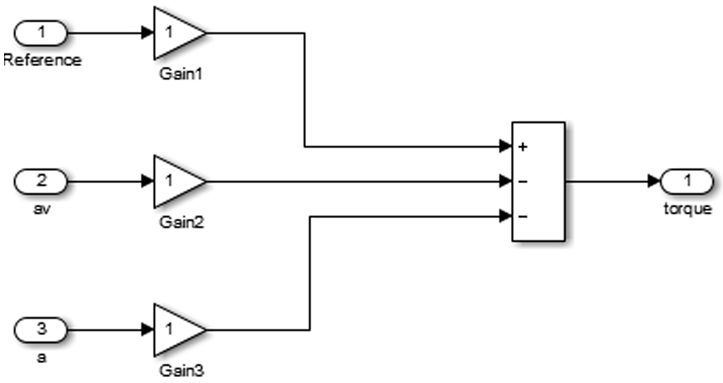
(b) SpaceEx Top Component

Fig. 36. Comparision between the two mains.

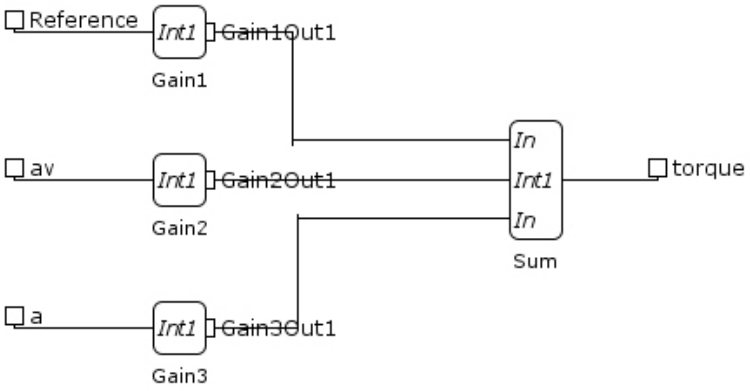
As you can see, the two models have the same graphical shape (position and dimension of the blocks, line-connections shape), and the same variables and associated names.

We have to check that also the hierarchical structure is preserved. By Figure 6.1 (resp., Figure 6.1) is possible to see that also this requirement is met:

the Simulink subsystem that models the controller (resp. the Plant) and the corresponding SpaceEx component have exactly the same components and connectios.



(a) SL SubSystem: *Controller*

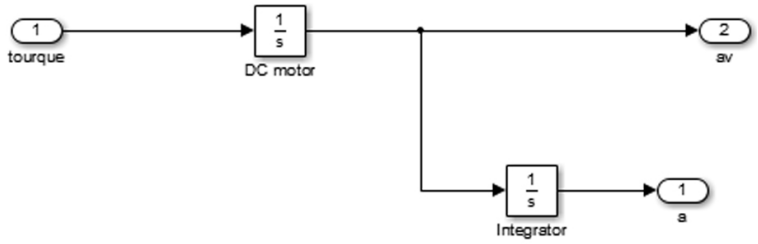


(b) SX Component: *Controller*

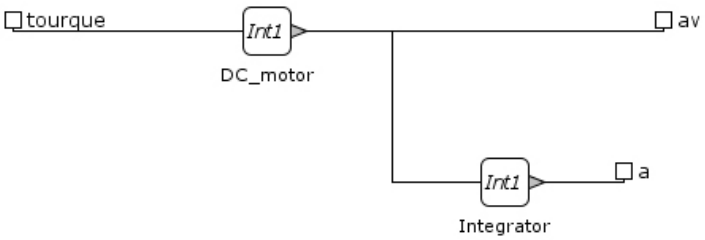
Fig. 37. Comparision between the two controllers.

Simulation Comparision In order to check if our translation is effectivly equivalent, we compare the result of the Simulink simulation on the model *DCMotor* with a simulation time equal to 50 seconds and the SpaceEx simulation on the model *SX_DCMotor*, with the same time-horizon (50 sec).

By Figure 6.1 is possible to see that we obtain the same result from the two simulations, starting from the initial state where all the variables are set to be equal to 0.



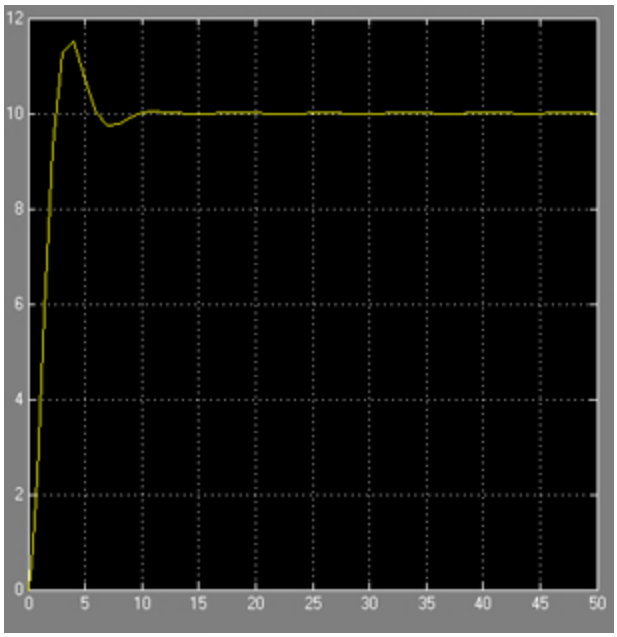
(a) SL SubSystem: *Plant*



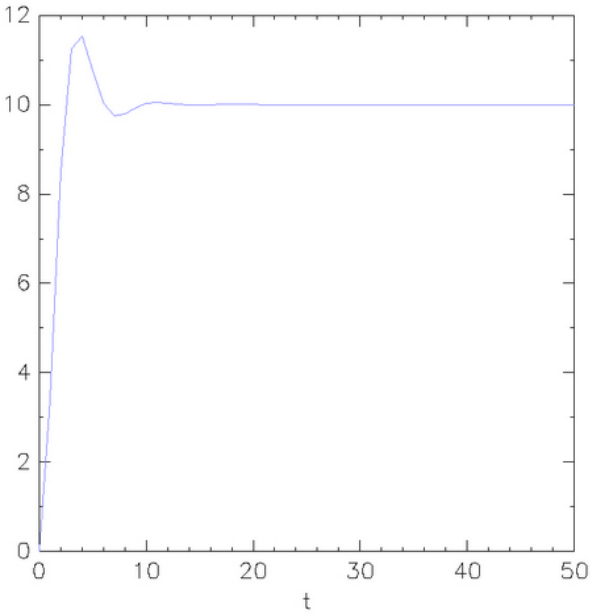
(b) SX Component: *Plant*

Fig. 38. Comparision between the two plants.

Formal Verification Figure 6.1 shows the result of the simulation over a time-horizon of 50 seconds and starting from the initial state where all the system variables are equal to zero. Now, suppose that we want to investigate about the behaviour of the plant-controller system when initial state is such that the second variable (av)) may be inside the interval $[0, 10]$: by Simulink, the only way to do this, is to choose some points p inside the interval $[0, 10]$ and then, for each of them, to perform a simulation where the initial state is such that $av = p$. By using this approach, to cover all the possible time evolution of the system inside the interval $[0, 10]$, one would need to perform an infinite number of simulations. Hence, the only thing that one can effectively do is to try to choose the highest possible number of points by following a "good" strategy, to try to cover as much as possible the allowed system trajectories. Clearly, this approach can not guarantee the correctness of the system under design. In the opposite, we can use SpaceEx to compute the set of **all** valuations reachable by the system, when the initial state is such that $0 \leq av \leq 10$. Figure 6.1 shows the results after 50 seconds. In order to compare the different potentially of the two approach (Formal Verification and Simulation), Figure 6.1 depicts the zoom over the first 5 seconds of both the results: notice that by the simulation is possible to obtain a single trajectory. Instead, by performing the reachability analysis, we obtain the entire (continuous) set of the states space.



(a) Simulink Simulation



(b) SpaceEx Simulation

Fig. 39. Comparision between the two simulations.

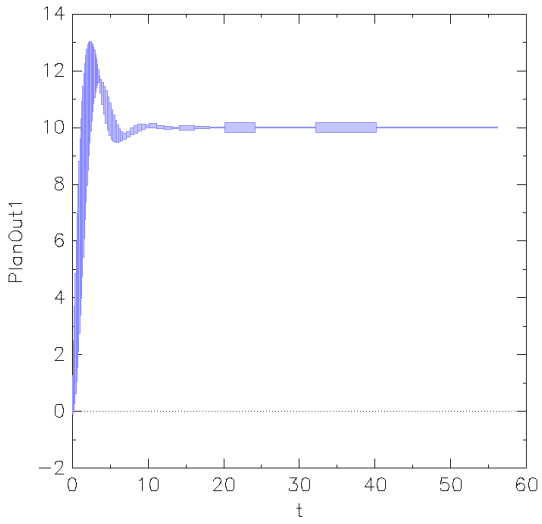
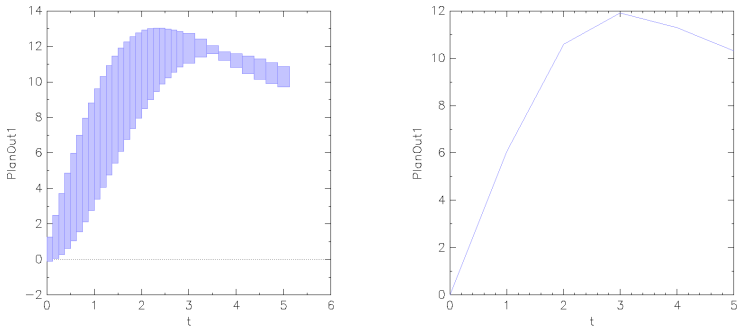


Fig. 40. The reachability set, when the initial state is a set of valuations.



(a) Formal Verification. (b) Simulation.

Fig. 41. Comparision over the first 5 seconds of evolution.

6.2 Semi-automatic Example: Foucault Pendulum

In this Section we will show how to translate a Simulink model of a Foucault pendulum. In particular here, after using the tool, the user needs to shortly manage the resulting SpaceEx model, in order to achieve the exact translation. The Simulink diagram that will be shown in this Section, is part of the Mathworks examples library, and it is used to model a Foucault pendulum (see `http://www.mathworks.com/help/simulink/examples/modeling-a-foucault-pendulum.html` for more details).

The Foucault pendulum was the brainchild of the French physicist Leon Foucault. It was intended to prove that Earth rotates around its axis. The oscillation plane of a Foucault pendulum rotates throughout the day as a result of axial rotation of the Earth. The plane of oscillation completes a whole circle in a time interval T, which depends on the geographical latitude. Foucault's most famous pendulum was installed inside the Paris Pantheon. This was a 28 kg metallic sphere attached to a 67 meter long wire. This example simulates a 67 meter long pendulum at the geographic latitude of Paris.

Figure 42 show the Simulink diagram used to model the Foucault pendulum.

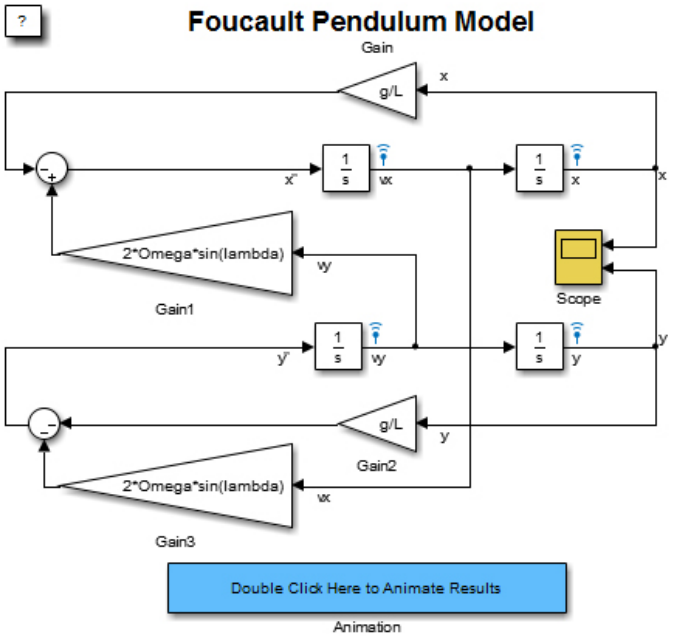


Fig. 42. SL diagram for Foucault pendulum.

In details, we have the following variables and parameters:

- $x, y =$ pendulum bob coordinates as seen by an observer on Earth.
- $Omega =$ earth's angular speed of revolution about its axis (rad/sec).
- $g =$ acceleration of gravity (m/sec^2).
- $L =$ length of the pendulum string (m).
- $lambda =$ geographic latitude (rad).

The initial conditions are defined as follows:

- $g = 9.83$.
- $L = 67$.
- $initial\_x = L/100$.
- $initial\_y = 0$.
- $initial\_xdot = 0$ (initial x velocity).
- $initial\_ydot = 0$ (initial y velocity).
- $Omega = 2 * pi/86400$.
- $lambda = 49/180 * pi$.

This model solves the differential equations for the Foucault Pendulum problem and plots the position of the pendulum bob in the XY plane. Notice that, the user can parametrize the system (g, $Omega$, $lambda$, L and the initial conditions) by using the MATLAB workspace, depicted in Figure 43

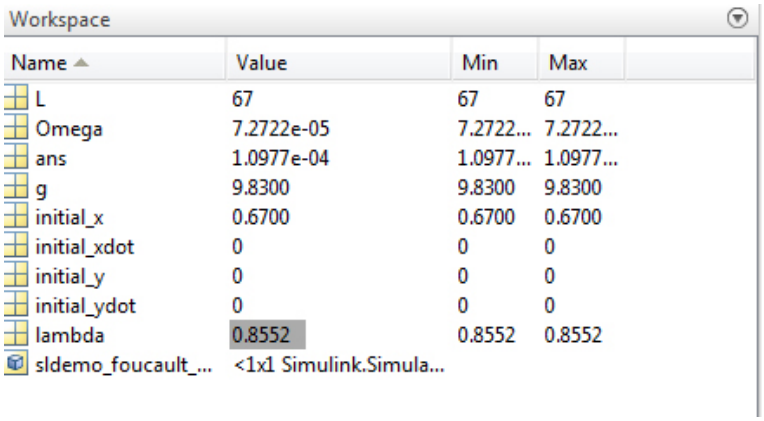


Fig. 43. Workspace associated to the example.

Similarly to the previous example, we need first to convert the .mdl file to the .xml format (by command *Save_System()*), and then to convert the obtained .xml file to the SpaceEx model by using the tool $SL2SX$. The resulting SpaceEx model is shown in Figure 44.

This model consists also in the network components $More\_Info$ and $Animation$ (as we said in the tool description, $SL2SX$ translates also simulink blocks that is not able to handle to give to the users the opportunity to manually complete

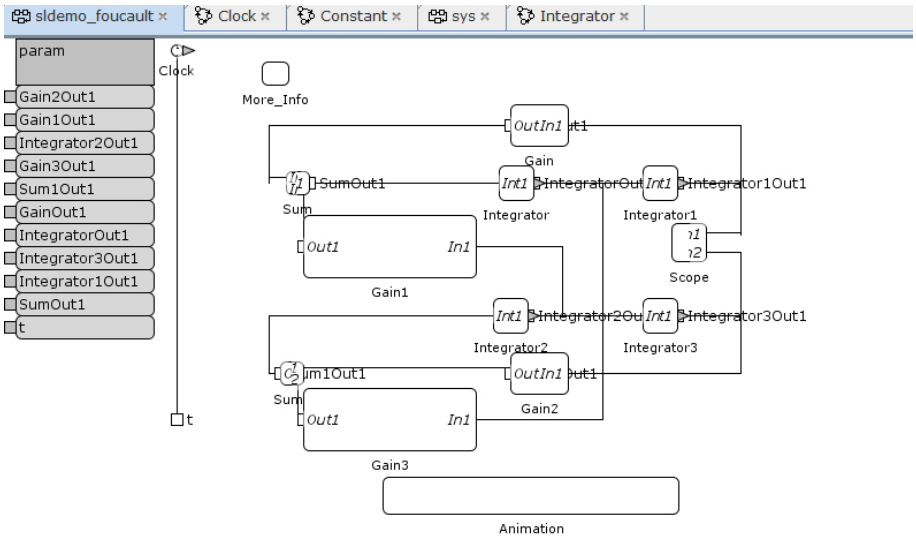


Fig. 44. SX model for Foucault Pendulum.

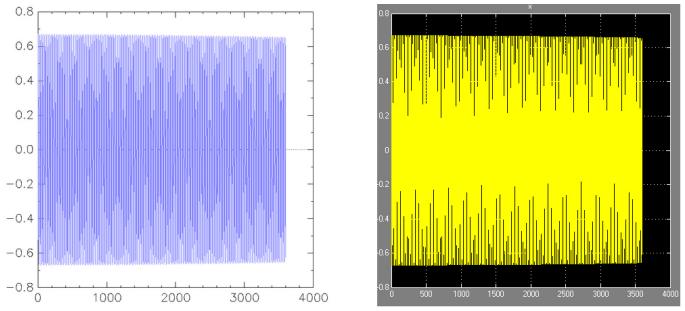
the translation). In this case is enough to just remove these networks, because they are useless for the simulation task (indeed, one is used to give a textual description of the model and the other one is used to show a graphical animation of the pendulum behavior).

Notice that, the components used to model the gain blocks, are mapped to the corresponding formula that express the gain (i.e. g/L and $2 * Omega * sin(lambda)$). But this way to express values is forbidden in SpaceEx. Hence, the user need to manually compute these values and then to make the map with them. In particular, we have:

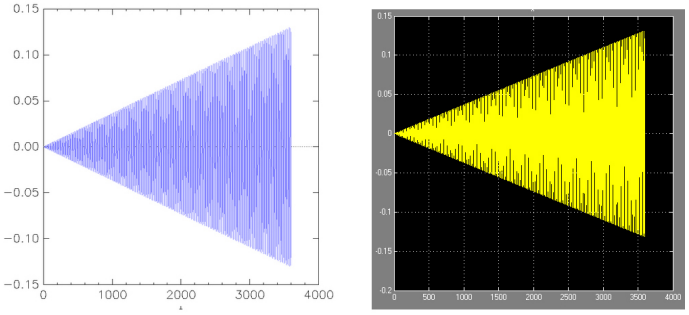
- $Gain$ and $Gain2$ associated to g/L. By looking at the workspace, we have $g/L = 0.980/67 = 0.1467$, and
- $Gain1$ and $Gain3$ associated to $2 * Omega * sin(lambda)$, that is $2 * Omega * sin(lambda) = 2 * 7.272205216643040e - 05 * sin(0.855211333477221) = 1.0977e - 04$.

By clicking on the respectively networks, the user can map each one to the computed value.

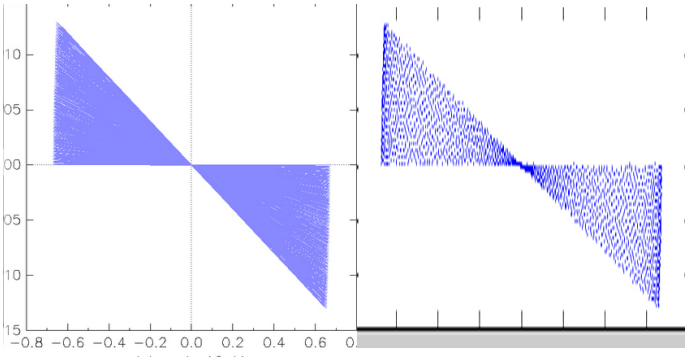
Simulation Before to launch the simulation, there is also another aspect to take into account: the workspace also define the initial conditions of the system. In particular, the initial value of x is 0.67 (i.e. $initial\_x = 0.67$). When we load the SpaceEx model, we have to initialize the variable $Integrator1Out1$ (that models the variable x) to the value 0.67 (this is done by the $Initialstate$ box, in the



(a) Plot of x position along 3600 sec.



(b) Plot of y position along 3600 sec.

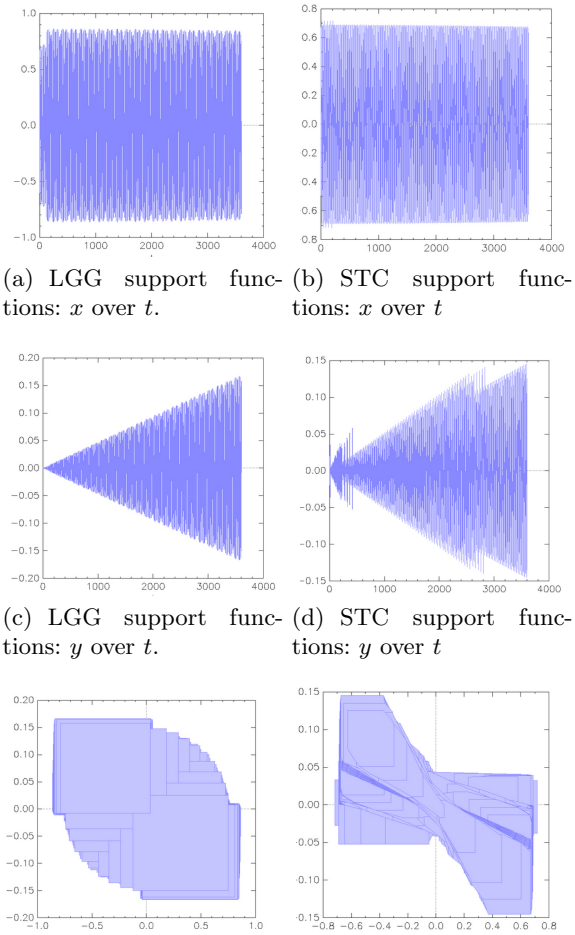


(c) Plot of x, y position after 3600 sec.

Fig. 45. SX (l.h.s.) and SL (r.h.s.) simulations for x and y coordinates over time.

Specification frame). After this, we are ready to launch the SpaceEx simulation and then to compare the result with the one of Simulink.

Figure 45 shows the the simulations result for x over time (Figure 45(a)), for y over time (Figure 45(b)) and for x, y (Figure 45(c)). We want to underline again that the results obtained by Simulink and SpaceEx are exactly the same.



(a) LGG support functions: x over t.

(b) STC support functions: x over t

(c) LGG support functions: y over t.

(d) STC support functions: y over t

(e) LGG support functions: x over y.

(f) STC support functions: x over y

Fig. 46. Performing SpaceEx verification on Foucault model, by using LGG (l.h.s.) and STC (r.h.s.) support functions.

Formal Verification In this paragraph, we will show the results of the verification when the initial conditions are such that x is inside the interval $[0.5, 0.67]$

(clearly, this kind of analisys is not possible by Simulink simulation). Figure 6.2 shows the results obtained by SpaceEx using two different kinds of state space representations (i.e. LGG and STC support functions).

6.3 Semi-automatic Example: Automotive Suspension

In this Section we will show how to translate a Simulink model of a simplified half car model that includes an independent front and rear vertical suspensions. Even in this case, the model is part of the Mathworks examples library (see `http://www.mathworks.com/help/simulink/examples/automotive-suspension.html` for more details).

The model also includes body pitch and bounce degrees of freedom. The example provides a description of the model to show how simulation can be used to investigate ride characteristics. You can use this model in conjunction with a powertrain simulation to investigate longitudinal shuffle resulting from changes in throttle setting.

Figure 47 show the Simulink diagram used to model the automotive suspension.

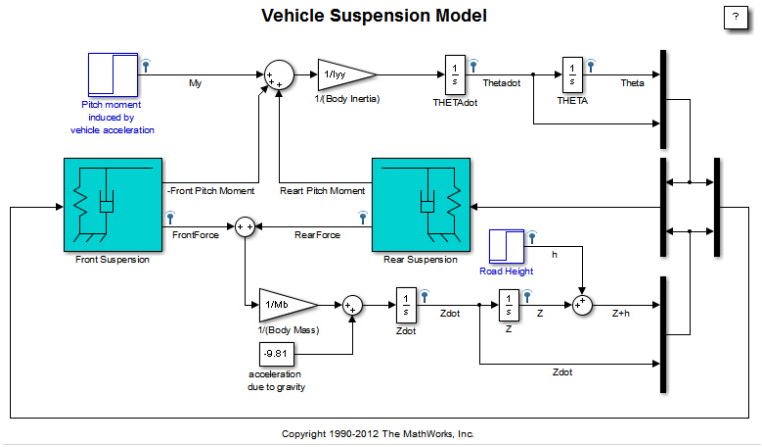


Fig. 47. SL diagram for automotive suspension.

The suspension model has two inputs, and both input blocks are blue on the model diagram. The first input is the road height. A step input here corresponds to the vehicle driving over a road surface with a step change in height. The second input is a horizontal force acting through the center of the wheels that results from braking or acceleration maneuvers. This input appears only as a moment about the pitch axis because the longitudinal body motion is not modeled.

The spring/damper subsystem that models the front and rear suspensions is shown in Figure 48.

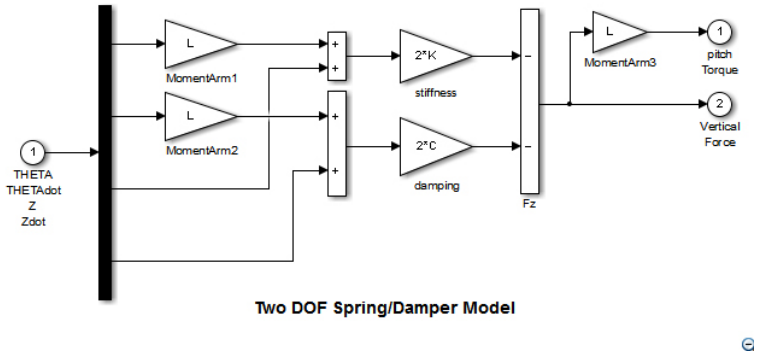


Fig. 48. Subsystem to model front and rear suspensions.

The default initial conditions are given as below:

- $Lf = 0.9$, front hub displacement from body gravity center (m).
- $Lr = 1.2$, rear hub displacement from body gravity center (m).
- $Mb = 1200$, body mass (kg).
- $Iyy = 2100$, body moment of inertia about y-axis in (kgm^2).
- $kf = 28000$, front suspension stiffness in (N/m).
- $kr = 21000$, rear suspension stiffness in (N/m).
- $cf = 2500$, front suspension damping in (N sec/m).
- $cr = 2000$, rear suspension damping in (N sec/m).

After converting the Simulink diagram to SpaceEx model by using $SL2SF$, we obtain the automata network depicts in Figure 49.

Similarly to the previous example, this model consists also in the network component $More\_Info1$, that clearly is useless for the simulation and verification tasks. Hence, the first manual interaction is to remove this network. Second, we need to map the gain blocks with the corresponding values coming from expressions used in the Simulink diagram, according to the initial conditions defined above. The last manual task, is to manage blocks that the current version of $SL2SF$ is not able to handle. In particular, we have to build an automaton to model the two Step blocks type and to correctly express the mux/demux blocks.

The Step used for the *Pitch moment by acceleration* is modeled by the automaton in Figure 50 (and the Step used for the *Road Height* is exactly the same but with different parameters on t and Out).

About the mux/demux, it is enough to delete them and then to modify the number of the input variables in *Front Suspension* and *Rear Suspension* subsystems according to the number of the elements of the mux. Then, we have to manually connect each of these inputs to the correct outputs, in particular (for both subsystems):

1. $In1$ mapped to $ThetaOut1$,

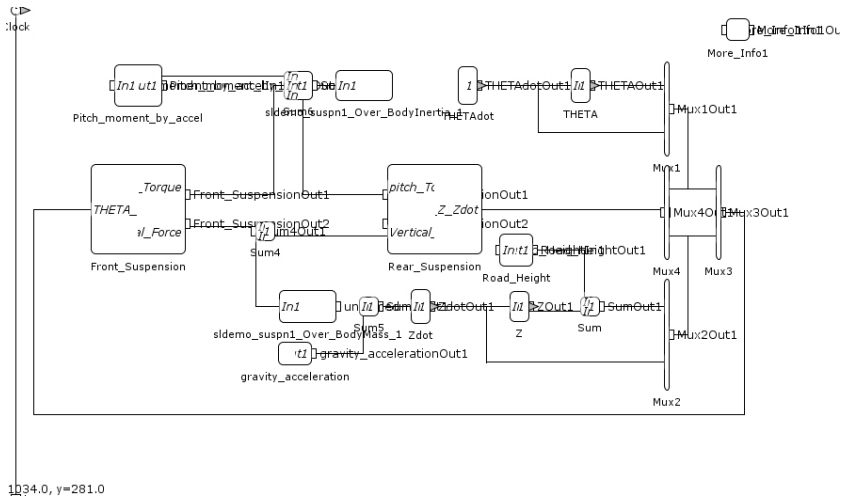


Fig. 49. SX model for automotive suspension.

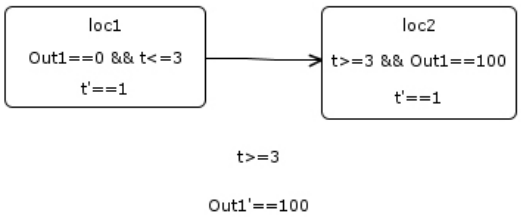


Fig. 50. Automaton to model the Step block type.

2. $In2$ mapped to $ThetadotOut1$,
3. $In3$ mapped to $ZOut1$,
4. $In4$ mapped to $ZdotOut1$.

After these manual operations, we obtain the SpaceEx model depicted in Figure 51.

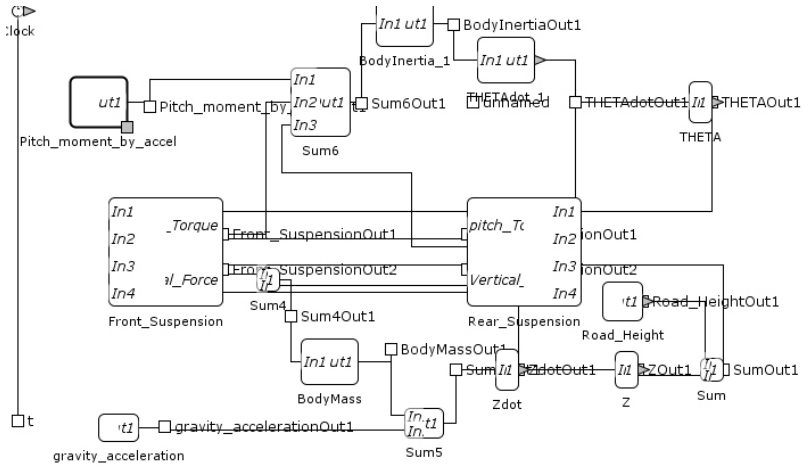


Fig. 51. SpaceEx model for automotive suspension ready to be simulated/verified.

SimulationZZZZZZZZZZZ Figure 52 shows the comparision between the two simulations (for Z over time $t = 3600$) and you can see that we obtain the same results.

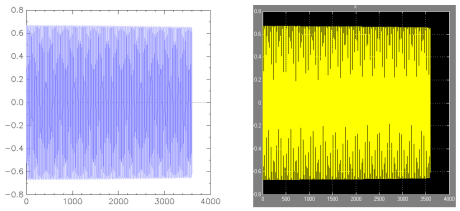
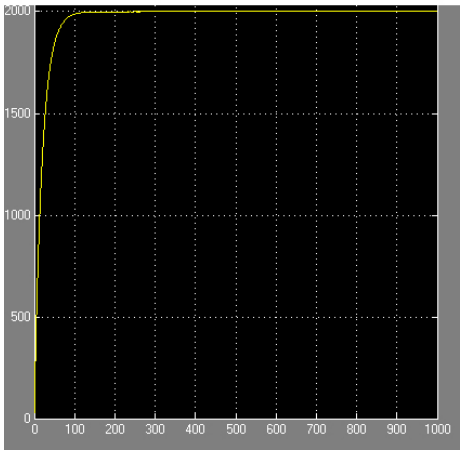


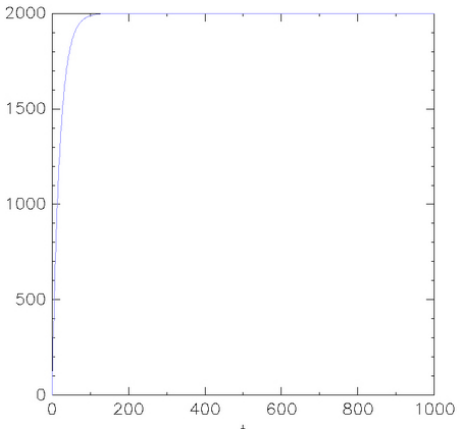
Fig. 52. SX (l.h.s.) and SL (r.h.s.) simulations for Z over time t.

7 External Test

In order to stress more our tool, we provided it to the Bosch Group and they make two tests. First of all, they replicated our first test case (see Section 6.1) and they confirmed our results. Second, and more important, they tested our tool on a fragment of one of their concrete industrial models. Because of the copyright, we can not show the Bosch model but we are able to present the result of the simulation that they obtained. In particular, Figure 7 depicts the comparison between the Simulink and SpaceEx simulations over 1000 sec, and again you can see that the result is the same.



(a) Simulink Simulation



(b) SpaceEx Simulation

Fig. 53. Comparision between the two simulations.

8 Requirements, Recommendations and Advanced Features

This Section will be a report about the current state (version 1.0) of the tool SL2SX, in terms of supported blocks, recommendations and bugs identified during the test phase.

8.1 Supported Blocks

Version 1.0 is able to process every Simulink block, meaning that for each of them, the translator builds the corresponding SpaceEx component or network. For the blocks that currently we are not able to give the equivalent hybrid automaton, the tool creates an "empty automaton" (and gives a warning) and leave the user free to manually implement the automaton. For the following blocks, the tool is already able to build the definitive, and equivalent, SpaceEx component:

1. Constant,
2. Gain,
3. Sum,
4. Product,
5. Integrator,
6. SubSystem,
7. Inport,
8. Outport.

Constant. A Simulink Constant Block is defined by the associated constant value K, and consists in just a single output. When such a block is meet, the tool builds an automaton whose name is the same of the block. The automaton consists in a single location, one variable ($Out1$) and one constant (K). The invariant of the location is set to be $Out1 == K$, where K is the constant value of the block. In this way, $Out1$ must assume always the value K, and hence the automaton is equivalent to the constant block.

Gain. A Gain block type consists in one input, one output and a constant value (the *gain* K) that is the gain from the input to the output. Clearly this means that the output will be equal K time the input. The tool translates this block by an automaton with one location, two variables In_1 and Out_1, and a constant K. The invariant of the location is set to be $Out_1 == K * In_1$, and hence the automaton is equivalent to the gain block.

Sum. A Sum block type consists in n input value, one output value and n symbols op_i (where $op_i \in \{+, -\}$), that describes the order of the sums to performs. For example, a sum block with two inputs and the symbols $+$, $-$, identifies a function that for output gives the first input minus the second input. The tool translates this block by an automaton with one location, and $n + 1$

variables $(In_1, \ldots, In_n, Out_1)$. The invariant of the location is set to be $Out_1 == (op_1)In_1(op_2)In_2 \ldots (op_n)In_n$, and hence the automaton is equivalent to the sum block.

Product. Similarly to the last case, a Product block type consists in n input value, one output value and n symbols op_i (where $op_i \in \{*, /\}$), that describes the order of the sums to performs. For example, a product block with two inputs and the symbols $*$, $*$, identifies a function that for output gives the first input times the second input. The tool translates this block by an automaton with one location, and $n+1$ variables $(In_1, \ldots, In_n, Out_1)$. The invariant of the location is set to be $Out_1 == (op_1)In_1(op_2)In_2 \ldots (op_n)In_n$, and hence the automaton is equivalent to the product block.

Integrator. The integrator block type consists in one input and one output, whose output is the derivative of the input. Hence, to translate it, the tool builds an automaton with one location and two variables (In_1 and Out_1). The location flow is set to be equal to $Out'_1 == In_1$, and clearly is equivalent to the Integrator block type.

Subsystem The tool translates a Subsystem block type by creating a new network component, with the same name of the subsystem, and will fill it with all the automata (already built) that are the translation of the equivalent block inside the subsystem.

Inport and Outport Finally, inports and outports block type are simply translated by adding, to the SpaceEx network component corresponding to the subsystem where that ports are defined, a corresponding variable, whose name is set to be equal to the In/Out port name.

8.2 Recommendations

When a user designs a Simulink diagram that is intended to be translated by SL2SX, is important to do not forgive the following constraints and recommendations:

1. The Simulink Block Mirroring is supported, but the related connection-lines could be not exactly the same as the SL source (only in the graphical shape).
2. Do not use the Matlab Workspace (it is not allowed to use, in the definition of the Block Parameter, a symbol defined in the workspace in place of the number)
   - As a consequence: do not use blocks that calls"OpenFnc" to initialize the system
3. The name of a block can not be defined in more text lines (i.e. do not use Carriage Return in the definition of a block name)
4. Avoid same name for different subsystems

SpaceEx Configuration File Once the SpaceEx model is obtained, the user has to complete the configuration file. The tool automatically writes it (with the same name of the SpaceEx model, but with *".cfg"* file extension), that contains the indication to perform simulation with duration equal to the *StopTime Simulation* defined in Simulink. Task of the user is to define the input state and the variables to look at.

8.3 Identified Bugs

There is a problem when the exported (by the Mathlab command "save_system") .xml file contains the special sequences of characters "&gt", "&lt", and similaria, used to express the single characters ">", "<", and others: it is necessary to edit the .xml to replace each of these occurrences with the corresponding character.

8.4 Advanced Features (inside the source code)

1. It is possible to change the level of the console verbosity during the translation, by the three variables *PrintSystemInfo* (if false, no info are given), *PrintBlockInfo* (if true, every time a Simulink Block is processed, the related info are displayed), *PrintVarsInfo* (if true, every time a SpaceEx variable is created, the related info are displayed). The variables are defined in the class *SLContentHandler*.
2. It is possible to change the (graphically) left-upper corner of the resulting spaceex model, by changing the value associated to the variable *hshift* in the class *SLContentHandler*. This value is important to avoid that inside the SpaceEx Editor, the list of variables overlaps part of the graphical model.

9 Future Extensions

This Section is a short summary for the future improvements of the tool. Same of them are already in progress.

1. Supporting other block type: switch, relay, deadzone and all the basic logic blocks (and, or, not, ...),
2. Supporting more complex dynamics (we are currently working to solve the problem on urgency on affine hybrid automata),
3. Supporting wireless signals,
4. Adding the automatically generation of the initial states,
5. Adding the autamatically generation of the output variables to show,
6. Complete the automatic generation of the SpaceEx configuration file.

Bibliography

[1] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. Softw. Eng.*, 22:181–201, March 1996.

[2] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[3] A. Balluchi, L. Benvenuti, T. Villa, H. Wong-Toi, and A. Sangiovanni-Vincentelli. Controller synthesis for hybrid systems with a lower bound on event separation. *Int. J. of Control*, 76(12):1171–1200, 2003.

[4] Nanette Bauer, Stefan Kowalewski, Guido Sand, and Thomas Löhl. A case study: Multi product batch plant for the demonstration of control and scheduling problems. In Sebastian Engell, Stefan Kowalewski, and Janan Zaytoon, editors, *ADPM'00*, pages 383–388. Shaker, 2000.

[5] D.A. Beek, M.A., Reniers, R.R.H., Schiffelers, and J.E. Rooda. Foundations of a compositional interchange format for hybrid systems. In *HSCC'07*, volume 4416 of *LNCS*, pages 587–600. Springer, 2007.

[6] G. Behrmann, A. David, and K. Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

[7] M. Benerecetti, M. Faella, and S. Minopoli. Automatic synthesis of switching controllers for linear hybrid systems: Safety control. *TCS: Theoretical Computer Science*, 493:116–138, 2012.

[8] Joseph T Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. *Ptolemy: A framework for simulating and prototyping heterogeneous systems*. Ablex Publishing Corporation, 1994.

[9] Christopher Chase, Joseph Serrano, and Peter J Ramadge. Periodicity and chaos from switched flow systems: contrasting examples of discretely controlled continuous systems. *Automatic Control, IEEE Transactions on*, 38(1):70–83, 1993.

[10] Pierre Collet and Jean Pierre Eckmann. *Iterated maps on the interval as dynamical systems*, volume 1. Springer, 1980.

[11] Alexandre Donzé and Goran Frehse. Modular, hierarchical models of control systems in spaceex. In *Control Conference (ECC), 2013 European*, pages 4244–4251. IEEE, 2013.

[12] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *CAV 11: Proc. of 23rd Conf. on Computer Aided Verification*, pages 379–395, 2011.

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

[14] B. Gebremichael and Frits F. Vaandrager. Specifying urgency in timed i/o automata. In *IEEE Int. Conf. Software Engineering and Formal Methods*, SEFM '05, pages 64–74, 2005.

[15] T. A. Henzinger, Pei-Hsin Ho, and H. Wong-Toi. Hytech: the next generation. In *Proc. IEEE Real-Time Systems Symposium (RTSS '95)*, page 56. IEEE Computer Society, 1995.

[16] T.A. Henzinger. The theory of hybrid automata. In *11th IEEE Symp. Logic in Comp. Sci.*, pages 278–292, 1996.

[17] Pei-Hsin Ho. *Automatic Analysis of Hybrid Systems*. PhD thesis, Cornell University, August 1995. Technical Report CSD-TR95-1536.

[18] MathWorks. Mathworks stateflow: Design and simulate state machines, September 2012. `http://www.mathworks.fr/products/simulink/`.

[19] MathWorks. Mathworks simulink: Simulation et model-based design, March 2014. `www.mathworks.fr/products/simulink`.

[20] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501–510, 1998.

[21] Stefano Minopoli and Goran Frehse. Non-convex invariants and urgency conditions on linear hybrid automata. In Axel Legay and Marius Bozga, editors, *Formal Modeling and Analysis of Timed Systems*, volume 8711 of *Lecture Notes in Computer Science*, pages 176–190. Springer International Publishing, 2014.

[22] Stefano Minopoli and Goran Frehse. Non-convex invariants and urgency conditions on linear hybrid automata. Technical Report TR-2014-4, Verimag, April 2014. `www-verimag.imag.fr`.

[23] Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems*, pages 149–178. Springer, 1993.

[24] JPM Schmitz, DA Van Beek, and JE Rooda. Chaos in discrete production systems? *Journal of Manufacturing Systems*, 21(3):236–246, 2002.

[25] C.J. Tomlin, J. Lygeros, and S. Shankar Sastry. A game theoretic approach to controller design for hybrid systems. *Proc. IEEE*, 88(7):949–970, 2000.

[26] H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *IEEE Conf. Decision and Control*, pages 4607 – 4612. IEEE, 1997.

[27] Martin Wulf, Laurent Doyen, and Jean-Franois Raskin. Almost asap semantics: From timed models to timed implementations. In *HSCC'04*, volume 2993 of *LNCS*, pages 296–310. Springer, 2004.