MULTIFORM

Integrated Multi-formalism Tool Support
for the Design of Networked Embedded Control Systems

Grant Agreement: FP7-ICT-2007-224249

Translation between CIF and SpaceEx/PHAVer

Deliverable No: 1.3.1

Due date: **01.03.2011**
Submission date: **25.05.2011**

Start date of the project: 01/09/2008 Duration: 45 months

Organization name of lead contractor for this deliverable: Université Joseph Fourier

Authors: Manish Goyal, Goran Frehse

Project co-funded by the European Commission within the Seventh Framework Programme (2007 - 2013)

Dissemination Level

PU	Public	x
PP	Restricted to other programme participants (incl. the Commission Services)	☐
RE	Restricted to a group specified by the consortium (incl. the Commission Services)	☐
CO	Confidential, only for members of the consortium (incl. the Commission Services)	☐

Translation between CIF and SpaceEx/PHAVer

Executive Summary:

This report describes the translation processes implemented in SpaceEx/PHAVer to read and write CIF models.

The Compositional Interchange Format (CIF) is a language framework for the exchange of continuous and hybrid models between different tools. SpaceEx/PHAVer is a verification platform for hybrid Systems. Based on a widely used formalism called hybrid automata, it aims to verify the given model of a hybrid system for its safety properties. SpaceEx's modeling languague, called SX, has the purpose to allow the exchange of models with a graphical user interface and model editor.

The objective of this report is to show current developments in the direction of bi-directional transformation between CIF and SX formats. We explain our work done so far ranging from CIF parsing, storing the information in ASTs and finally, conversion to the SX format by reading this information. It is important to mention that SpaceEx directly parses the CIF format without any need of an external parser. CIF being a very rich format in terms of its semantics, we use only a subset of its features, those currently applicable and relevant in the very context of SpaceEx.

Three ways of translation are presented in this report: to read models in the CIF format in SpaceEx/PHAVer, to convert models in CIF format to the SpaceEx/PHAVer proprietary SX format, and to convert models from SX format to the CIF format. The translation results are illustrated with examples.

Translation between CIF and SpaceEx/PHAVer

as part of

MULTIFORM
Integrated Multi-formalism Tool Support
for the Design of Networked Embedded Control Systems

by

Manish Goyal

under the guidance of

Dr. Goran Frehse

Verimag Research Lab
Grenoble, France

May 2011

Contents

List of Figures

1 Introduction

Because of their vast applicability across various domains, *control systems* are studied, explored and employed for analyzing real life problems in many fields, including industrial applications. However, the difficulty in analyzing them because of their behavior and scalability always motivates the researchers to design new analysis tools and improve existing ones. *Model-based design* is one of the most important and popular design-paradigms based on which such tools are developed. The technique offers increased safety, better and easy to understand design and therefore, reduced effort in designing these systems.

We have tools like Matlab/Simulink, Modelica [8], gPROMS [9], UPPAAL [7], PHAVer [4] and SpaceEx [6], to name a few. Nevertheless, these tools differ in the design formalisms and have their own strengths and weeknesses. It points to the need of a framework that should integrate all these tools and in turn, employs useful features of each and every tool leaving aside its complexities. The framework must also automate the transformations among tools with different formalisms and thus, act as an interconnection between them. But, this framework can only be realized through a generic relied upon intermediate format that should assist the transformations between a large number of models. The goal of Multiform is to develope a model exchange framework supported by the *Compositional Interchange Format* (CIF) [10].

SpaceEx is a verification platform for a class of dynamical systems known as *Hybrid Systems.* Based on a widely used formalism called hybrid automata, it aims to verify the given model of a hybrid system for its safety properties. SpaceEx's modeling languague is called SX [2], whose purpose is to allow the exchange of models with a graphical user interface and model editor. PHAVer [4] is another verification tool for linear hybrid automata, which is now included in SpaceEx.

The objective of this report is to show current developments in the direction of bi-directional transformation between CIF and SX formats. We explain our work done so far ranging from CIF parsing, storing the information in ASTs and finally, conversion to the SX format by reading this infomation. It is important to mention that SpaceEx directly parses the CIF format without any need of an external parser. CIF being a very rich format in terms of its semantics, we use only a subset of its features, those currently applicable and relevant in the very context of SpaceEx.

The report is structured as follows. Section 2 presents the SpaceEx internal data structures and the CIF format. We discuss the subset of the CIF grammar that is successfully parsed by SpaceEx followed by corresponding railroad diagrams. In Section 3, we first discuss the SpaceEx modeling language i.e., SX and its constructs. Later, we show an SX component transformed from its CIF counterpart via SpaceEx internal representation. The transformation from an SX component to the CIF model is demonstrated in Section 4. In Section 5, we talk about the work yet to be performed in this direction.

2 Reading CIF in SpaceEx

The *Compositional Interchange Format* (CIF) [10] is the intermediate format for the Multiform model exchange framework. It is used to establish inter-operability of a wide range of tool formalisms by means of model transformations. It provides a generic model exchange formalism since it encompasses various language concepts that are present in modern modeling formalisms. In our case, the use of CIF is also motivated by the fact that both CIF and SpaceEx are automaton-based languages, which greatly simplifies the transformation.

2.1 SpaceEx Internal Data Structures

SpaceEx [5, 6] is a platform for the verification of hybrid systems. It allows one to model hybrid systems and then verify these models for their safety properties, using its reachability algorithm. It uses an internal representation for its various model constructs (Figure 1(c)). We discuss, in brief, these data structures below. However, the reader is encouraged to refer [5] for further details.

1. An *automaton* consists of a graph in which each vertex, called *location*, has a *flow* and an *invariant.* The edges in this graph are known as *transitions.* Each transition is represented with its *guard* and the *assignment.*
2. *Automata Network* is shown as the parallel composition of one or more automata.
3. The *variables* and the *labels* are global (or, have unique names).
4. The automata obtain unique names with proper instantiation rules and the bindings.

2.2 Internal CIF parser of SpaceEx

SpaceEx parses CIF into various meta structures (*network*, *automaton* etc.) represented by *abstract syntax trees* (ASTs) that are then transformed into corresponding structures of the final SpaceEx model. We present various grammar constructs and their mappings from CIF to those in SpaceEx (cf. Fig. 1). Fig. 1(d) depicts the *edge semantics* of Fig. 1 which means, a different edge type represents a different relation between two grammar-constructs. For e.g, An edge with a (non-solid) diamond on one end stands for the *"is a part of"* relation. Therefore, the AutomatonDefinition *is a part of* Model in the CIF grammar. On the same note, an *Edge* in the AST structure *"is mapped to"* a *transition* of the SpaceEx model. And in the SpaceEx model, a *transition "is associated with"* a *source* as well as a *target* location.

2.2.1 Parser Grammar

In this section, the CIF grammar is defined with the rules for its structures. Each definition is explained and accompanied by an example for better understanding. It captures a subset of the original CIF grammar that is significant in our case. Moreover, we have also made an attempt to correlate our definitions to those in the original grammar, wherever possible. The name of a grammar construct defined by SpaceEx for parsing CIF, is represented in *italics* with its first letter capital. Whereas, the name written in sans serif shows a definition-name in the CIF grammar [10].

1. *Network* : Currently, we deal with only one network component where a *Network* is expressed as a Model with either of below 2 definitions (cf. Figure 2):
   - One or more automata are first instantiated in the network definition and defined later. The network definition consists of a *network name*, *parameters declarations*, *Automata_ decs* followed by *Automata_ defs*.

```
model PushButton_Lamp()=
|[ act PushButton1On
,      PushButton1Off
,      Lamp1On
,      Lamp1Off
:: button1 : Button (PushButton1On, PushButton1Off)
|| lamp1 : Lamp (Lamp1On, Lamp1Off)
|| Controller (PushButton1On, PushButton1Off, Lamp1On, Lamp1Off)
|| user1 : User (PushButton1On, PushButton1Off) ]|
```

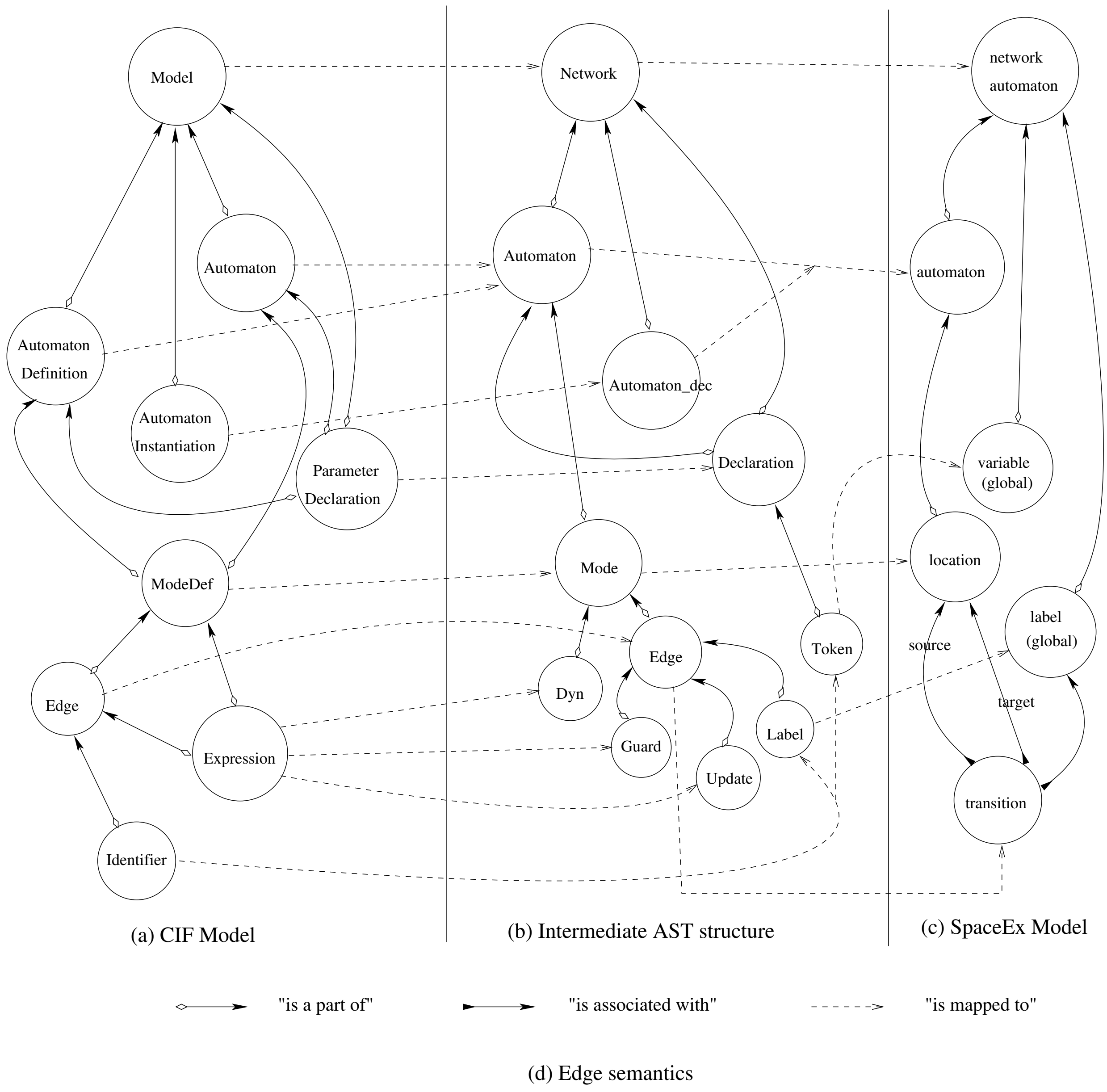


Figure 1: CIF Model to SpaceEx Model

```
automaton Button(inout act sync ButtonOn, ButtonOff) =
|( mode released = initial
                         (act ButtonOn) goto pushed
,          pushed   = (act ButtonOff) goto released
)|

automaton Controller(inout act sync ButtonOn, ButtonOff, LampOn, LampOff) =
|( mode released = initial
                         (act ButtonOn) goto pushed
                         (act LampOff) goto released
,          pushed   = (act ButtonOff) goto released
                         (act LampOn) goto pushed
)|

automaton Lamp(inout act sync LampOn, LampOff) =
|( mode off = initial
                    (act LampOn) goto on
,          on = (act LampOff) goto off
)|

automaton User(inout act sync ButtonOn, ButtonOff) =
|( clock control real $t = 0.0$
; mode off = initial
                    (when $t \geq 1.0$ now act ButtonOn do $t := 0.0$) goto on
,          on = (when $t \geq 2.0$ now act ButtonOff do $t := 0.0$) goto off
)|
```

- In another definition, the automata are defined within the network definition with *network name*, *parameters declarations* and *Automata.*

```
model TankController()=
|[ cont control real $V = 10.0$
; var                 real $Qi$, $Qo$
; disc control  nat $n = 0$
:: Tank : |( mode physics = initial
                           inv $V' = Qi - Qo$
                           ,    $Qi = n * 5.0$
                           ,    $Qo = V$
              )|

||

Controller : |( mode closed = initial
                                   (when $V \geq 2$ now do $n := 1$) goto opened
               ,           opened =(when $V \geq 10$ now do $n := 0$) goto closed
               )|
]|
```

2. *Automaton_dec* (Fig. 5(iv)) : It is equivalent to AutomatonInstantiation where an *instance name* (optional) is followed by the *automaton name* i.e., instance type and the expressions that are passed as arguments to the *Automaton.*

button1: Button(PushButton1On, PushButton1Off)

or,

Button(PushButton1On, PushButton1Off)

3. *Automaton* : An *Automaton* (Fig. 3) is either an AutomatonDefinition or an Automaton.

   - AutomatonDefinition has a *name* followed by *formal parameters declarations* (optional) and/or *local parameters declarations* and the *Modes*.

```
automaton User(inout act sync ButtonOn, ButtonOff) =
|( clock control real t = 0.0
; mode off = initial
                (when t ≥ 1.0 now act ButtonOn do t := 0.0) goto on
,         on = (when t ≥ 2.0 now act ButtonOff do t := 0.0) goto off
)|
```

   - An Automaton is expressed with its *name*, *local parameter declarations* (optional) and ',' seperated *Modes*.

```
Tank : |( mode physics = initial
                inv V' = Qi − Qo
                ,    Qi = n ∗ 5.0
                ,    Qo = V
       )|
```

4. *Mode* (Fig. 4) is ModeDef having a *name*, keyword **initial** (optional), its dynamics (*Dyns*) and the *Edges*.

```
mode off = initial
        (when t ≥ 1.0 now act ButtonOn do t := 0.0) goto on
```

5. The *Dyns* (Fig. 6) are defined with a keyword (**inv** or **flow**)[1] and one or more ',' seperated *Dyn's* (Expressions).

```
inv V' = Qi − Qo
, Qi = n ∗ 5.0
, Qo = V
```

6. An *Edge* (Fig. 7) or Edge can have a *Guard* and/or, a *label* (**act** and the *label* name), *Update* and in the end, *goto* statement (**goto** and the *target location* name).

```
(when t ≥ 1.0 now act ButtonOn do t := 0.0) goto on
or,
(when t ≥ ttr now do (VB, g) := (0, 1)) goto filling
```

[1]SpaceEx distinguishes between an invariant and flow so that even if the invariant is not empty in a location, the flow can be empty i.e., $flow = \phi$, which means time can't pass in that location.

or,

(**now do** $x := 1$) **goto** pushed

7. *Guard* (Fig. 8(i)) is represented with the keyword **when** followed by an Expression.

   when $V \geq 2$

8. *Update* (Fig. 8(i)) is shown as keyword **do** and an Expression.

   do $t := 0.0$

9. The *Declarations* are seperated by ';' (cf. Figure 10(i)).

   cont control real $V = 10.0$
   ; **var** **real** Qi, Qo

10. A *Declaration* (Fig. 10(ii)) is either a formal parameters' (non-local) declaration or the local parameters' declaration.

    - A *formal parameters' declaration* is consisting of *Param_ type*, *control* (optional), *Static_ type* and ',' seperated *Dec_ expressions.*

      inout act sync ButtonOn, ButtonOff

    - On the other hand, a *local parameters' declaration* is formed of *Param_ type*, *Static_ type* (optional), *sync* (optional) and *Dec_ expressions.*

      clock control real $t = 0.0$

11. A *Dec_ expression* (Fig. 11) is either just an *identifier* or an *identifier* (parameter name) followed by its *Value* where, *Value* is either an *int* or *double.*

12. Scoped variables are assigned unique names with a prefix notation.

2.2.2 Railroad Diagrams

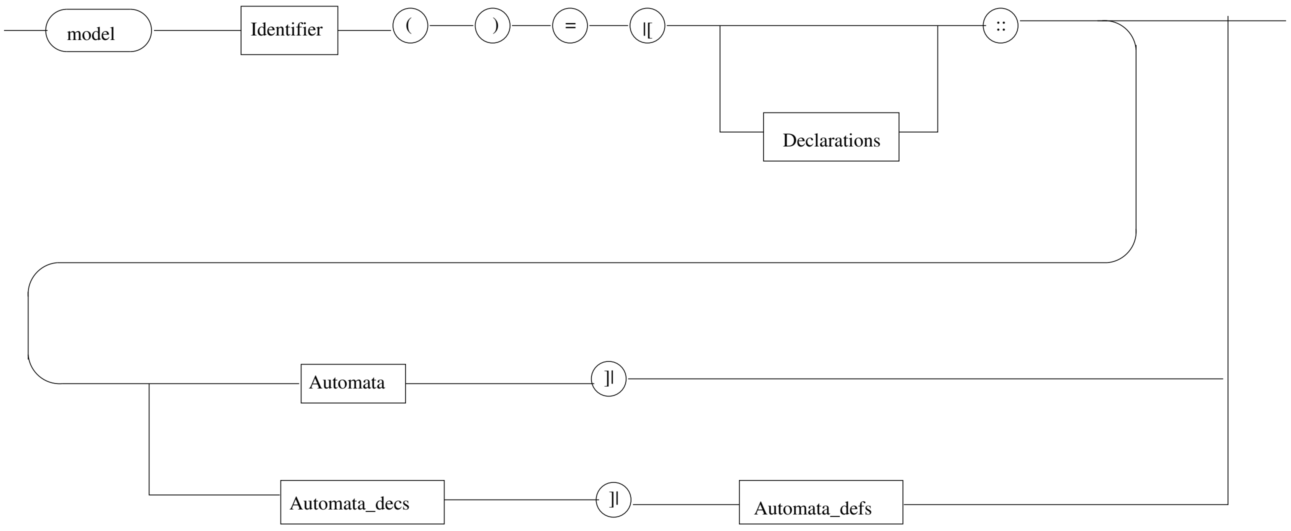


Figure 2: Network

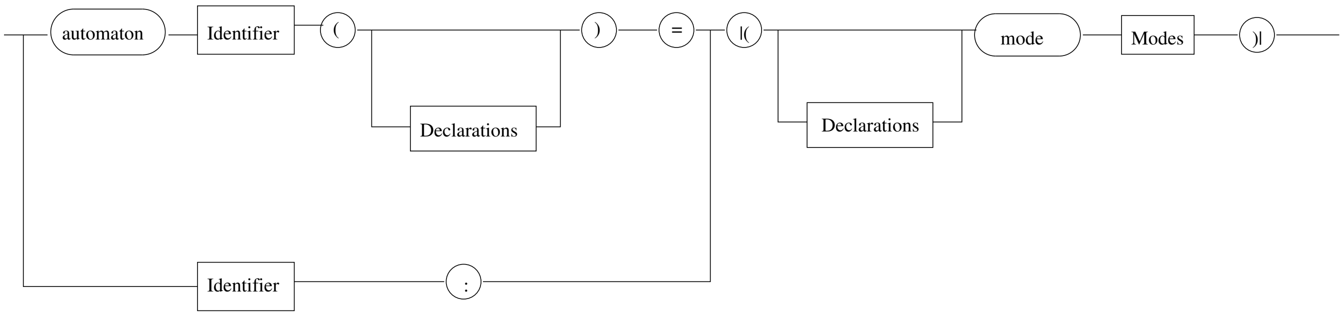
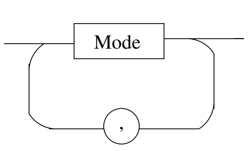
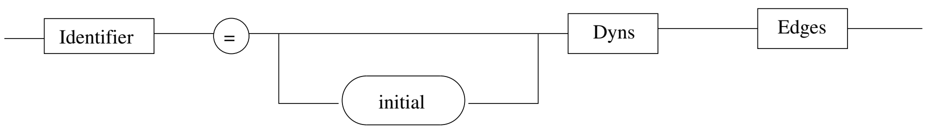


Figure 3: Automaton

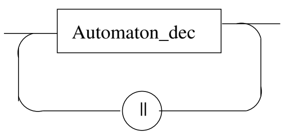


(i) Modes

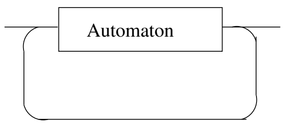


(ii) Mode

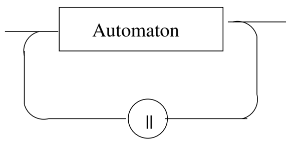
Figure 4: Modes



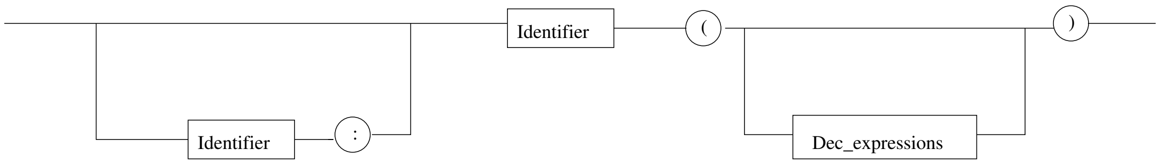
(i) Automata_decs



(ii) Automata_defs

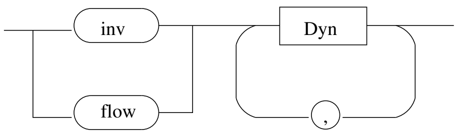


(iii) Automata

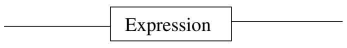


(iv) Automaton_dec

Figure 5: Automata and their Instantiations

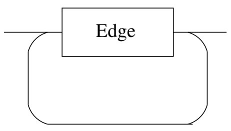


(i) Dyns

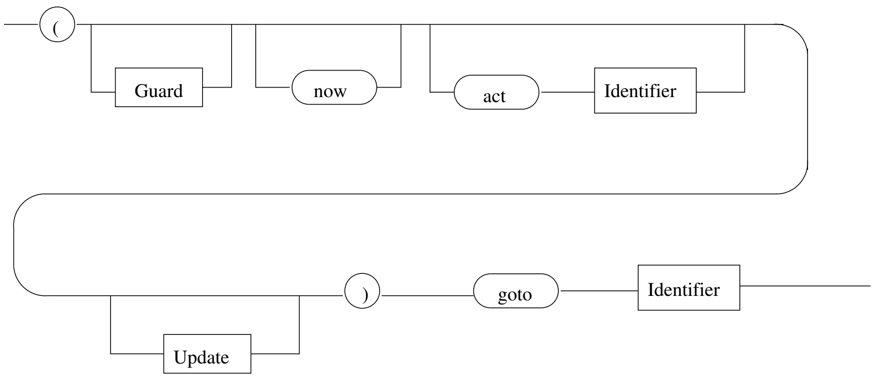


(ii) Dyn

Figure 6: Dynamics

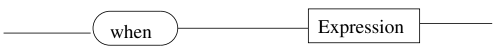


(i) Edges

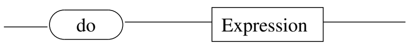


(ii) Edge

Figure 7: Edges

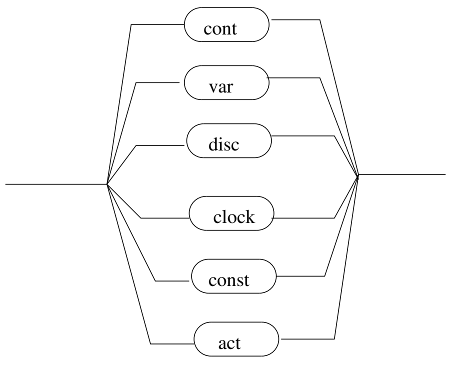


(i) Guard

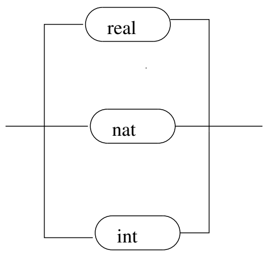


(ii) Update

Figure 8: Guard and Update

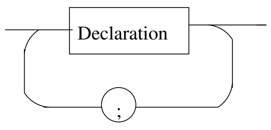


(i) Param_type

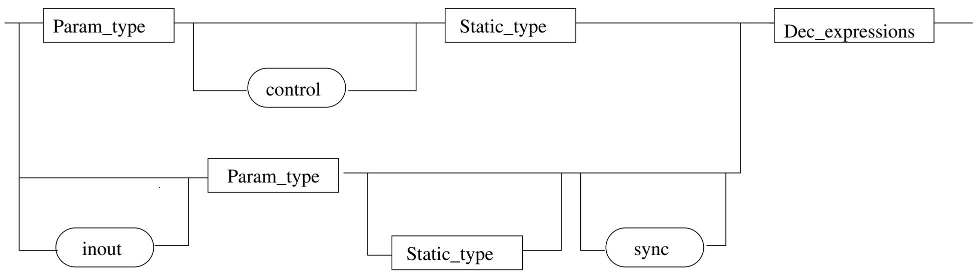


(ii) Static_type

Figure 9: Types

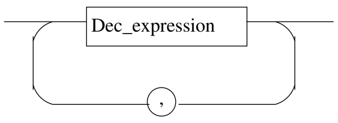


(i) Declarations

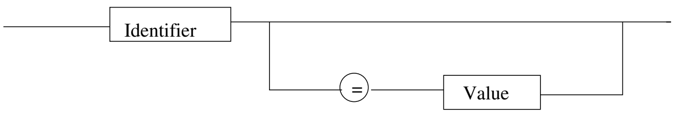


(ii) Declaration

Figure 10: Declarations



(i) Dec_expressions



(ii) Dec_expression

Figure 11: Declaration Expressions

3 Transforming CIF to SX Format

3.1 Modeling in SX Format

SpaceEx's modeling language, called SX [2], is an XML based format supported by a graphical model editor. SX Models are similar to the hybrid automata known in literature [1], except that they provide a richer mechanism of hierarchy, templates and instantiations. An model consists of one or more components. When SpaceEx reads a model file, it translates the components into either an automaton or into a network of automata in parallel composition. So, for the purposes of analysis, a component defines a hybrid automaton, with everything else (hierarchy etc.) being syntactic sugar whose only purpose is making the construction of complex models easier.

3.1.1 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 defined 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 constant, which is treated as a variable with constant dynamics, i.e., it does not change its value over time.[1] It may be attributed a value, like 9.81, when it is instantiated in a network component.
- a synchronization label.

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

3.1.2 Base Components

A base component in the model is translated by SpaceEx into a *hybrid automaton* [1]. It consists of a graph in which each vertex, called a *location*, is associated with a *flow*, which is a set of differential equations (or inclusions) that defines 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* (sometimes called a *reset* or a *discrete jump*). 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 satisfies 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 from 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 flow of a location is a set of differential equations. The type of constraints allowed on the derivatives depends on the scenario. The LGG scenario accepts non-deterministic affine dynamics of the form

$$\dot{x} = Ax + Bu + b_0, \tag{1}$$

[1] A variable with constant dynamics is commonly referred to as a "parameter" of a hybrid automaton, but these should not be confused with the formal parameters of components.

where $\dot{x}$ is the derivative of x with respect to time and u is a set of non-deterministic inputs. Constraints on u can be integrated in the invariant. The model format does not currently support vector or matrix notation. A system of the above form is described by the expression

```
x1' == a11*x1 + ... + b11*u1 + ... +b01 &
xn' == an1*x1 + ... + bn1*u1 + ... +b0n
```

where the prime behind a variable denotes its derivative.

Similarly, the discrete jump associated with a transition modifies the variables with an assignment of the form

$$x := Rx + Su + s_0,$$

where u is a set of non-deterministic inputs that is given by constraints in the invariant of the source location of the transition. The assignment can be described in two forms, either as

```
x1 := r11*x1 + ... + s11*u1 + ... &
xn := rn1*x1 + ... + sn1*u1 + ...
```

or in a general relation form as linear constraints over x and x', where the prime denotes the value after the transition.

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 unsatisfiable constraint by `false`.

3.1.3 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 attribute values to their constants.

When the 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 fly*, so that only the reachable parts of that automaton are actually created in memory.

When instantiating a component H in network K, every formal parameter of H must be bound to either a (potentially local) formal parameter of K or to a numeric value.

Components inside the network K can be connected by binding their variables or synchronization labels to the same symbols in K. Any component can declare any variable as one of 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 in alternation. In certain application areas, like mechanics or electrical 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 trajectory after a certain point in time. We say that "time stops" in the model. This may or may not be a modeling error. In nondeterministic models, states in which time stops occur frequently as an artifact of over-approximating complex dynamics with simpler ones.

There are no inputs and outputs in SpaceEx models, but are *controlled* and *uncontrolled variables*, which are used in compositional reasoning and are somewhat related [3]. Simply put, a controlled variable

x cannot be modified outside of the component that "owns" it beyond what is possible in the component itself. SpaceEx enforces these semantics by modifying every component J in which x is uncontrolled as follows:

- For any transition of J that does not synchronize with a transition in H, add the assignment $x' == x$.
- To all locations q of J, add a self-loop that nondeterministically resets x to any value in the invariant of J.

Users who wish to apply compositional reasoning need to declare every variable x as controlled in at most one base component H, and as uncontrolled in all other base components J. Users who do not care about compositional reasoning can simply declare all variables as controlled in all components.

Controlled variables are also used by SpaceEx in transition assignments, simply to make the notation of transitions easier for the user: Unless otherwise specified, a controlled variable remains constant in a transition. I.e., if the transition assignment does not mention a controlled variable x, SpaceEx adds the assignment $x' == x$. In cases where this is not desired, e.g., for algebraic variables (variables whose derivatives are not defined), one can declare the variable as uncontrolled.

3.1.4 Initial and Forbidden States

The specification of a reachability problem includes the set of initial states, from which all behaviors of the system originate. 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

$$\texttt{loc(} < \textit{component} > \texttt{)==} < \textit{location} >,$$
$$\texttt{loc(} < \textit{component} > \texttt{)!=} < \textit{location} >,$$

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

3.2 Mapping CIF to SX

The SX format, being modular, captures the models in a user- and GUI-friendly manner. We show, for the purpose of illustration, the mapping from various CIF constructs to those in an SX model directly (Figure 12). The *PushButton_ Lamp* CIF model from Section 2 is considered for the discussion.

1. A CIF Model corresponds to a network component in SX which is represented by the parallel composition of its components. For e.g., the CIF model

```
model PushButton_ Lamp()=
|[ act PushButton1On
,      PushButton1Off
,      Lamp1On
,      Lamp1Off
:: button1 : Button (PushButton1On, PushButton1Off)
|| lamp1 : Lamp (Lamp1On, Lamp1Off)
|| Controller (PushButton1On, PushButton1Off, Lamp1On, Lamp1Off)
|| user1 : User (PushButton1On, PushButton1Off)
]|
```

corresponds to the following network component in the SX:

```
<component id= "PushButton_Lamp.button1~PushButton_Lamp.lamp1
                          ~PushButton_Lamp.Controller~PushButton_Lamp.user1">
         <param name="PushButton_Lamp.user1.t" type="real" d1="1" d2="1"
                          local="false" dynamics="any" controlled="true"/>
         <param name="PushButton_Lamp.PushButton1Off" type="label" local="false"/>
         <param name="PushButton_Lamp.PushButton1On" type="label" local="false"/>
         <param name="PushButton_Lamp.Lamp1Off" type="label" local="false"/>
         <param name="PushButton_Lamp.Lamp1On" type="label" local="false"/>
               ...
</component>.
```

2. An Automaton in the CIF corresponds to a base component in the SX. For example, the *User* automaton in the CIF with instantiation *user1*

```
user1 : User (PushButton1On, PushButton1Off)

automaton User(inout act sync ButtonOn, ButtonOff) =
|( clock control real t = 0.0
; mode off = initial
                (when t ≥ 1.0 now act ButtonOn do t := 0.0) goto on
,          on = (when t ≥ 2.0 now act ButtonOff do t := 0.0) goto off
)|
```

is transformed into the *PushButton_ Lamp.user1* component.

```
<component id="PushButton_Lamp.user1">
         <param name="t" type="real" d1="1" d2="1" local="false" dynamics="any" controlled="true"/>
         <param name="PushButton_Lamp.PushButton1Off" type="label" local="false"/>
         <param name="PushButton_Lamp.PushButton1On" type="label" local="false"/>
         <location id="1" name="off">
               <invariant> true </invariant>
               <flow> t' == 1 </flow>
         </location>
         <location id="2" name="on">
               <invariant> true </invariant>
               <flow> t' == 1 </flow>
         </location>
         <transition source="1" target="2">
               <label>PushButton_Lamp.PushButton1On</label>
               <guard> t &gt;= 1 </guard>
               <assignment> t' == 0 </assignment>
         </transition>
         <transition source="2" target="1">
               <label>PushButton_Lamp.PushButton1Off</label>
               <guard> t &gt;= 2 </guard>
               <assignment> t' == 0 </assignment>
         </transition>
</component>
```

3. The CIF ModeDef is mapped to a location in the SX component.
4. The Edge in a CIF ModeDef is represented by a transition in the SX component.
5. The invariant, flow and guard in the SX model are mapped from their counterparts in the CIF.
6. For each clock variable, t, in a component, the predicate $t' == 1$ is added to the flow in every location of that component.
7. The identifier following the **act** keyword in a CIF mode corresponds to the label name in a SX component.
8. The assignment tag in a transition refers to the reset map. For each control variable, x, absent in the assignment predicate of a component, a constraint $x' == x$ is added to the assignment predicate. This is done to enforce that the values for these variables should not change, unless specified, over the transition.

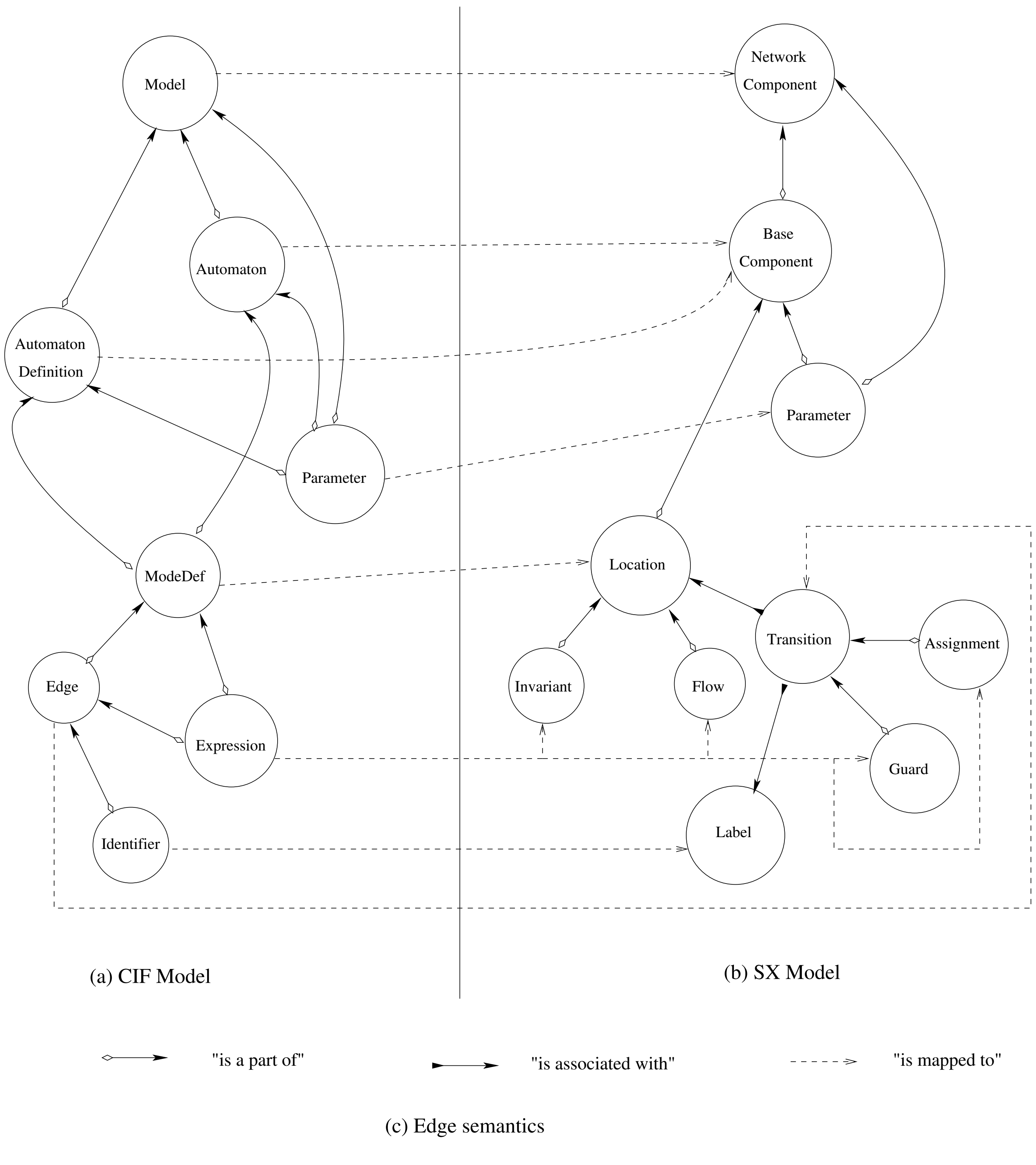


Figure 12: Mapping CIF Model to SX Model

Finally, the *PushButton_ Lamp* network component in SX can be represented as follows:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<sspaceex xmlns="http://www-verimag.imag.fr/xml-namespaces/sspaceex" version="0.2" math="SpaceEx">
<component id= "PushButton_Lamp.button1~PushButton_Lamp.lamp1
                        ~PushButton_Lamp.Controller~PushButton_Lamp.user1">
      <param name="PushButton_Lamp.user1.t" type="real" d1="1" d2="1"
                        local="false" dynamics="any" controlled="true"/>
      <param name="PushButton_Lamp.PushButton1Off" type="label" local="false"/>
      <param name="PushButton_Lamp.PushButton1On" type="label" local="false"/>
      <param name="PushButton_Lamp.Lamp1Off" type="label" local="false"/>
      <param name="PushButton_Lamp.Lamp1On" type="label" local="false"/>

      <location id="1" name="released~off~released~off">
          <invariant> true </invariant>
          <flow>PushButton_Lamp.user1.t' == 1 </flow>
      </location>
          ...
      <location id="5" name="released~on~released~off">
          <invariant> true </invariant>
          <flow>PushButton_Lamp.user1.t' == 1 </flow>
      </location>
          ...
      <location id="12" name="pushed~off~pushed~on">
          <invariant> true </invariant>
          <flow>PushButton_Lamp.user1.t' == 1 </flow>
      </location>
          ...
      <location id="16" name="pushed~on~pushed~on">
          <invariant> true </invariant>
          <flow>PushButton_Lamp.user1.t' == 1 </flow>
      </location>

      <transition source="1" target="12">
          <label>PushButton_Lamp.PushButton1On</label>
          <guard> PushButton_Lamp.user1.t &gt; = 1 </guard>
          <assignment> PushButton_Lamp.user1.t' == 0 </assignment>
      </transition>
      <transition source="5" target="16">
          <label>PushButton_Lamp.PushButton1On</label>
          <guard> PushButton_Lamp.user1.t &gt; = 1 </guard>
          <assignment> PushButton_Lamp.user1.t' == 0 </assignment>
      </transition>
          ...
      <transition source="16" target="5">
          <label>PushButton_Lamp.PushButton1Off</label>
          <guard> PushButton_Lamp.user1.t &gt; = 2 </guard>
          <assignment> PushButton_Lamp.user1.t' == 0 </assignment>
      </transition>
</component>
</sspaceex>.
```

3.3 CIF to SX via SpaceEx Internal Data Structures

In our implementation, the actual transformation from a CIF model to the SX model takes place through SpaceEx internal data structures as shown in Figure 13. As discussed in Section 2, SpaceEx first parses and maps a CIF model to the SpaceEx model. Then, it transforms this SpaceEx model into an SX model (cf. Figure 14).

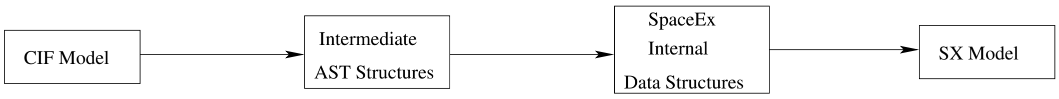


Figure 13: CIF to SX via SpaceEx Internal Data Structures

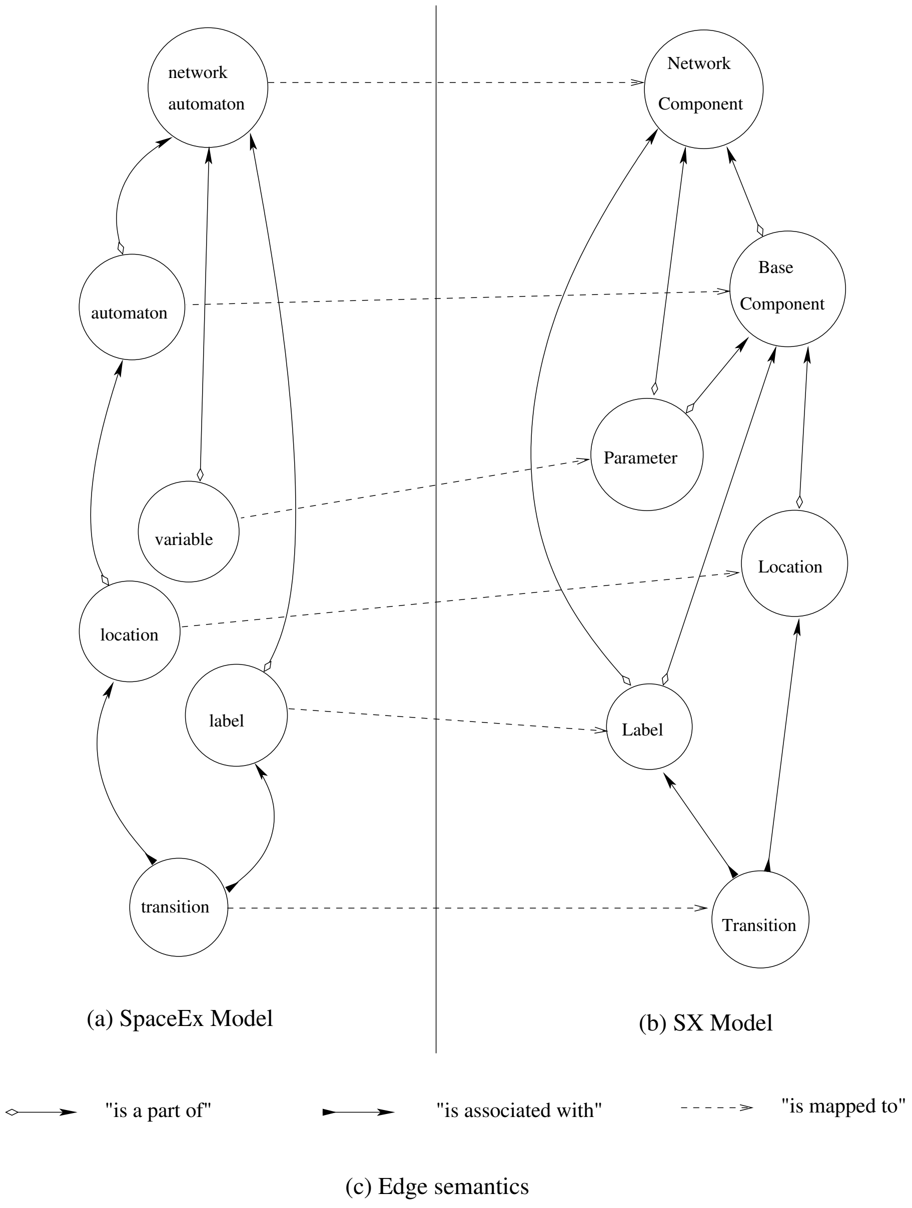


Figure 14: SpaceEx Model to SX Model

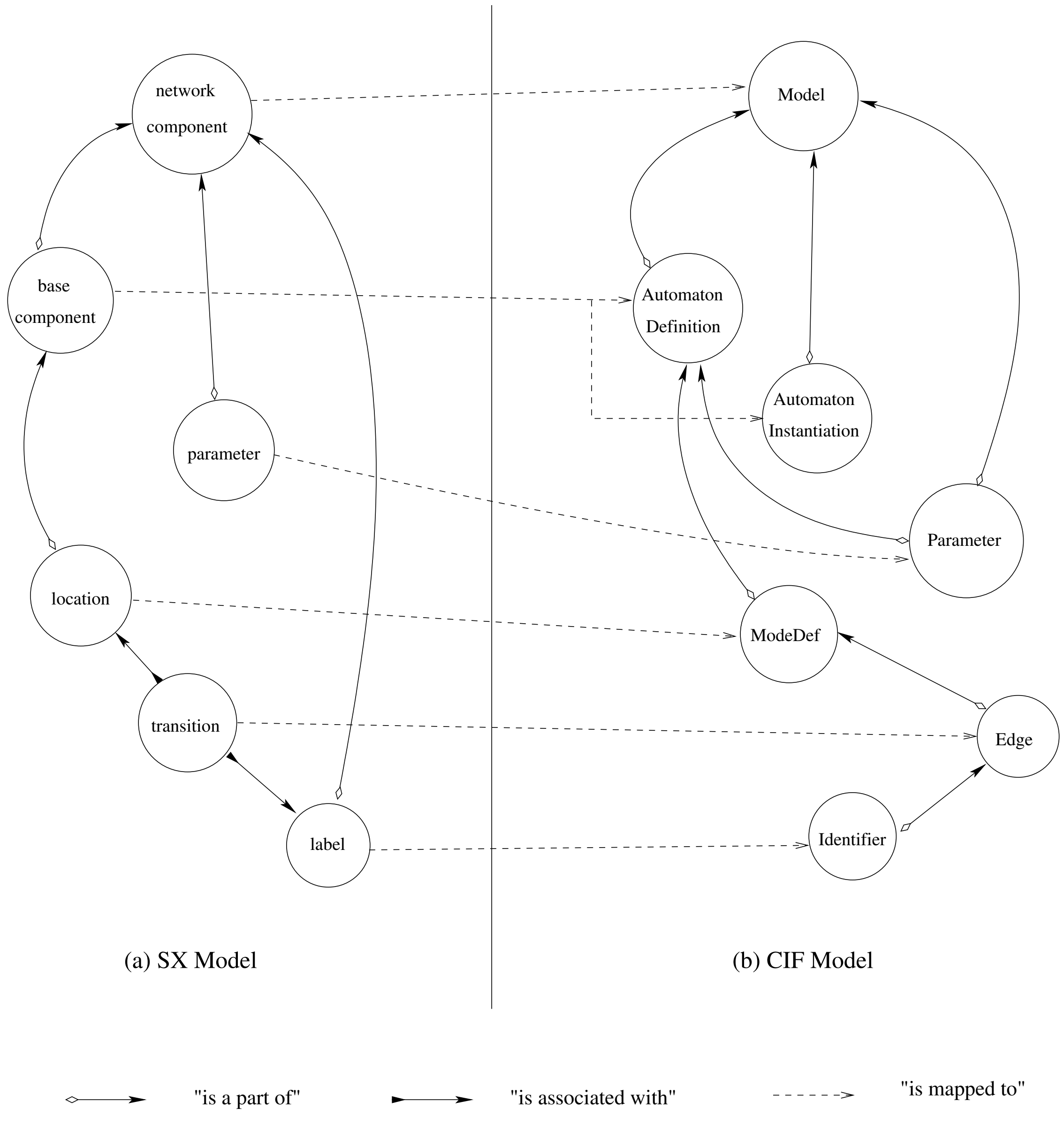


Figure 15: Transforming SX to CIF

4 Transforming SX to CIF

SX to CIF transformation is reverse of that discussed in Section 3.3. In this section, we talk about the mapping from SX to CIF constructs, the notational aspect adapted while transformation, and the restrictions posed due to SpaceEx internal representation. *PushButton_ Lamp* component from Section 3.2 is considered for the demonstration.

Mapping The *network component* in SX is mapped to the CIF Model. Each *base component* is mapped to an AutomatonInstantiation and AutomatonDefinition in the CIF model as shown in Figure 15. For e.g., *PushButton_ Lamp.user1* base component is instantiated and defined as follows:

```
PushButton_Lamp.user1 (PushButton_Lamp.user1.t,PushButton_Lamp.PushButton1Off,
                                                 PushButton_Lamp.PushButton1On)
```

```
automaton PushButton_Lamp.user1(var PushButton_Lamp.user1.t;
                                inout act sync PushButton_Lamp.PushButton1Off;
                                inout act sync PushButton_Lamp.PushButton1On) =
|( mode off = flow PushButton_Lamp.user1.t' = 1
             (when PushButton_Lamp.user1.t >= 1 now
             act PushButton_Lamp.PushButton1On do PushButton_Lamp.user1.t' := 0) goto on
,       on = flow PushButton_Lamp.user1.t' = 1
             (when PushButton_Lamp.user1.t >= 2 now
             act PushButton_Lamp.PushButton1Off do PushButton_Lamp.user1.t' := 0) goto off
)|
```

Similarly, the parameters in an SX component are mapped to the model parameters as well as the formal parameters in the CIF model.

Notation In the original CIF model, the parameter declarations of one or more parameters of the same type, are seperated by ','. On the other hand, the declarations of different types of parameters are delimeted by ';'. For simplicity, however, SpaceEx does not distinguish between these two forms of parameter declarations, during transformation into CIF format. It means that, in the transformed CIF model, parameter declarations are seperated by ';' irrespective of the parameter types. Importantly, the notation obtained this way still conforms to the CIF format. For instance,

```
  act PushButton_Lamp.PushButton1Off, PushButton_Lamp.PushButton1On
, PushButton_Lamp.Lamp1Off, PushButton_Lamp.Lamp1On
```

in the original CIF model is represented in the transformed model as

```
  act PushButton_Lamp.PushButton1Off; act PushButton_Lamp.PushButton1On
; act PushButton_Lamp.Lamp1Off; act PushButton_Lamp.Lamp1On
```

While conjunctions are donated with an *ampersand*(&) in SX, they are denoted with a *comma*(,) in CIF.

Restrictions Since the transformation in our implementation takes place through SpaceEx internal representation, SpaceEx loses some information pertaining to the variable scope/context and automaton template. Therefore, the variables and labels are used with their global names in the transformed CIF model. Similarly, SpaceEx treats each base component as a seperate automaton such that every template instantiation becomes a separate automaton. It leads to as many AutomatonDefinitions as the number of AutomatonInstantiations in our transformed CIF model.

The CIF *PushButton_ Lamp* Model shown in Section 2.2, read in SpaceEx and then backtransformed to CIF, yields the following result:

```
model PushButton_Lamp()=
|[ var PushButton_Lamp.user1.t
;  act PushButton_Lamp.PushButton1Off
;  act PushButton_Lamp.PushButton1On
;  act PushButton_Lamp.Lamp1Off
;  act PushButton_Lamp.Lamp1On
:: PushButton_Lamp.button1 (PushButton_Lamp.PushButton1Off, PushButton_Lamp.PushButton1On)
|| PushButton_Lamp.lamp1 (PushButton_Lamp.Lamp1Off, PushButton_Lamp.Lamp1On)
|| PushButton_Lamp.Controller ( PushButton_Lamp.PushButton1Off, PushButton_Lamp.PushButton1On,
                                PushButton_Lamp.Lamp1Off, PushButton_Lamp.Lamp1On)
|| PushButton_Lamp.user1 (PushButton_Lamp.user1.t, PushButton_Lamp.PushButton1Off,
                          PushButton_Lamp.PushButton1On)
]|

automaton PushButton_Lamp.button1(inout act sync PushButton_Lamp.PushButton1Off;
                                  inout act sync PushButton_Lamp.Lamp1On) =
|( mode released = (act PushButton_Lamp.PushButton1On) goto pushed
,       pushed   = (act PushButton_Lamp.PushButton1Off) goto released
)|

automaton PushButton_Lamp.lamp1(inout act sync PushButton_Lamp.Lamp1Off;
                                inout act sync PushButton_Lamp.PushButton1On) =
|( mode off = (act PushButton_Lamp.Lamp1On) goto on
,       on = (act PushButton_Lamp.Lamp1Off) goto off
)|

automaton PushButton_Lamp.Controller(inout act sync PushButton_Lamp.PushButton1Off;
                                     inout act sync PushButton_Lamp.PushButton1On;
                                     inout act sync PushButton_Lamp.Lamp1Off;
                                     inout act sync PushButton_Lamp.Lamp1On) =
|( mode released = (act PushButton_Lamp.PushButton1On) goto pushed
                   (act PushButton_Lamp.Lamp1Off) goto released
,       pushed   = (act PushButton_Lamp.PushButton1Off) goto released
                   (act PushButton_Lamp.Lamp1On) goto pushed
)|

automaton PushButton_Lamp.user1(var PushButton_Lamp.user1.t;
                                inout act sync PushButton_Lamp.PushButton1Off;
                                inout act sync PushButton_Lamp.PushButton1On) =
|( mode off = flow PushButton_Lamp.user1.t' = 1
              (when PushButton_Lamp.user1.t >= 1 now
              act PushButton_Lamp.PushButton1On do PushButton_Lamp.user1.t' := 0) goto on
,       on = flow PushButton_Lamp.user1.t' = 1
              (when PushButton_Lamp.user1.t >= 2 now
              act PushButton_Lamp.PushButton1Off do PushButton_Lamp.user1.t' := 0) goto off
)|
```

5 Future work

SpaceEx can, so far, parse only the linear functions in the CIF. Therefore, the support for parsing non-linear functions by the SpaceEx is targetted for the implementation during the current phase of the project.

References

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

[2] Scott Cotton, Goran Frehse, and Olivier Lebeltel. The SpaceEx Modeling Language: A Technical Report. `http://spaceex.imag.fr/documentation/user-documentation/`, Dec 2010.

[3] Goran Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. PhD thesis, Radboud Universiteit Nijmegen, October 2005.

[4] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems past HyTech. *STTT*, 10(3):263–279, 2008.

[5] Goran Frehse. An Introduction to SpaceEx v0.8: A Technical Report. `http://spaceex.imag.fr/documentation/user-documentation/`, Dec 2010.

[6] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable Verification of Hybrid Systems. In Shaz Qadeer Ganesh Gopalakrishnan, editor, *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[7] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[8] Modelica Association. Modelica - a unified object-oriented language for physical systems modeling. `https://www.modelica.org/association`, 2011.

[9] Process Systems Enterprise Ltd. (PSE). gPROMS. `http://www.psenterprise.com/gproms`, 2010.

[10] D. A. van Beek, Michel A. Reniers, Ramon R. H. Schiffelers, and J. E. Rooda. Foundations of a Compositional Interchange Format for Hybrid Systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 587–600. Springer, 2007.