The SpaceEx Modeling Language

Scott Cotton, Goran Frehse, Olivier Lebeltel

December 8, 2010

Abstract

This paper describes the modeling language used in SpaceEx. We provide two tutorial examples that illustrate how hybrid automata are defined and composed. A formal grammar is provided and semantic restrictions on the acceptable files are given.

1 Introduction

Hybrid automata are a widely used formalism for describing systems with both continuous and discrete dynamics. Their theory is well understood and they are a fairly compact formal description of the complex behavior of such systems. Powerful analysis algorithms are available and subject to active research. We have implemented hybrid automata and some of these algorithms in our verification tool SpaceEx/PHAVer [4]. In this paper we present its modeling language, called SX, whose purpose is to allow the exchange of models with a graphical user interface and model editor, as well as the exchange with other tools and modeling languages via automatic translation.

In practise, systems frequently composed of several interacting entities, which share information via continuous signals or quantities as well as discrete communication signals or shared events. To be usable and concise, the communication between these entities is often described in a hierarchical, nested fashion, with interfaces that abstract from local variables and events. We will call such models *modular*.

Classic definitions for hybrid automata and their parallel composition do not have local variables and events. However, it is possible to automatically translate modular models to flat hybrid automata. One of the purposes of the SX language is to capture modular models in a user- and GUI-friendly and then to allow SpaceEx to perform an automatic instantiation of a flat hybrid automaton representation that suits its algorithms.

When entities are structurally similar and vary only in their parameter values, models can be reused and often very complex models are constructed from a few basic building blocks. In addition to the advantages of model reuse, this can be relevant for translations to and from similar modular languages like the Simulink format of The MathWorks. For the translation of languages that share such a structure, it suffices to provide suitable models for a small set of basic building blocks and an automatic translation of one structure to the other in order to be able to translate complex models.

Syntactic sugar like constants and vector/matrix notations can be quite helpful in practice, so we have tried to include these as well.

Formally, the SX format is specied by a grammer in Compact RelaxNG format [5], which is enclosed in the appendix. A readable tutorial is available on the web [3], An

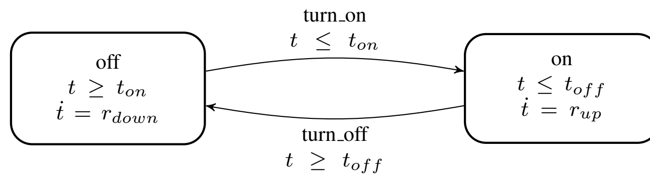


Figure 1: A simple hybrid automaton model of a room heater

automatic validator called Jing is publicly available [1], as well as a translator to other grammar formats [2].

In the next section we present the SX language features using two examples. Then we describe how the model gets instantiated to a set of hybrid automata and which restrictions we impose on the model in order to be able to do such an instantiation. A formal grammar and the complete example files can be found in the appendix.

2 Tutorial Examples

We present two examples in order to illustrate the basic concepts of the SX language: Declaration of local/uncontrolled/controlled variables, defining constants and composing components.

A formal description of the language elements will be given in the next section.

2.1 Heater Example

Consider the simple model of a room heater with thermostat shown in Fig. 1. The temperature t rises with rate $r_{up} > 0$ if the heater is on, and falls with rate $r_{down} < 0$ if the heater is off. The heater switches on and off at thresholds t_{off} and t_{on}. It can be verified that the heater keeps the temperature between t_{off} and t_{on}.

The hybrid automaton model has two locations, off and on. A transition with label turn_on can take the automaton from location off to location on if its *guard* $t \leq t_{on}$ is satisfied. The automaton doesn't have to follow this transition. Instead, it may remain in location off as long as its *invariant* $t \geq t_{on}$ is satisfied. In this model, the guard and invariant overlap at the point $t = t_{on}$, and it is clear that since $\dot{t} > 0$ in location off, the automaton has no choice but to take the transition immediately when this state is reached. Similarly, a transition with label turn_off takes the automaton from location on back to location off when the temperature gets above t_{off}.

The automaton corresponds in the SX format to a base component. We'll create two components: one in which the parameters r_{up} etc. are unbound, i.e., they do not yet have values, and one in which we bind the parameters to values so we can analyise the behavior for that specific instance.

We start with the generic version:

```
<component id="HeaterTemplate">
  ...
</component>
```

The variable t of the automaton is formal parameters of the component. Its value *type* is `real`, and since it's a scalar variable its dimensions are `d1="1"` and `d1="1"`. We want t to be in the interface of the component, so it's not local. We also don't need extra

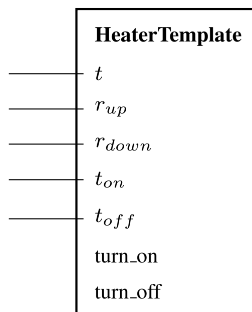


Figure 2: HeaterTemplate component

error checking on the dynamics – what we'll write is what we get, so we set *dynamics* to `any`:

```
<component id="HeaterTemplate">
  <param name="t" type="real" d1="1" d2="1" local="false" dynamics="any
      " />
  ...
</component>
```

The parameters $r_{up}, r_{down}, t_{on}, t_{off}$ are formal parameters that we need to declare for the component. They are real-valued and scalar, and as parameters of a hybrid automaton they do by definition not change their values during an execution of the automaton. In the SX format, this is called `const` dynamics. Since we wish to later bind them to constant values, they have to be in the interface of the HeaterTemplate component and cannot be local:

```
<param name="r_up"   type="real" d1="1" d2="1" local="false" dynamics="
    const" />
<param name="r_down" type="real" d1="1" d2="1" local="false" dynamics="
    const" />
<param name="t_on"   type="real" d1="1" d2="1" local="false" dynamics="
    const" />
<param name="t_off"  type="real" d1="1" d2="1" local="false" dynamics="
    const" />
```

Finally, we need to declare the synchronization labels turn_on and turn_off. If we wish them to synchronize with labels of other components, we need to declare them as part of the components interface, i.e., non-local. Here we don't consider the component to be connected to other components, so we let them be local, and they won't show in the interface of the component:

```
<param name="turn_on"  type="label" local="true"/>
<param name="turn_off" type="label" local="true"/>
```

The above declarations of the formal parameters fully defines the external interface of the component. This gives us all the information we need to know whether and how that component can be connected to other components, or whether and how its parameters can be bound to constant values. A schematic representation of the component is shown in Fig; 2.

The definition of the locations and transitions is fairly straightforward. We only need to introduce ids for the locations, since transitions are defined by the ids of their

source and target locations. The derivative of variables in the flow equations is denoted with a prime.

```
<location id="1" name="off"
  <invariant> t &gt;= t_on </invariant>
  <flow> t' == r_down </flow>
</location>
<location id="2">
  <invariant> t &lt;= t_off </invariant>
  <flow> t' == r_up </flow>
</location>
```

The change of variables in a discrete transition is denoted by an `assignment` tag containing an equation over the value before and the value after the transition, the latter being designated by a prime. The value does not change, which we can denote with the equation

```
  <assignment> t' == t </assignment>
```

For convenience, equations that express that a variable doesn't change can simply be omitted. In sum, the transitions are defined as follows:

```
<transition source="1" target="2">
  <label>turn_on</label>
  <guard> t &lt;= t_on </guard>
  <assignment> t' == t </assignment>
</transition>
<transition source="2" target="1">
  <label>turn_off</label>
  <guard> t &gt;= t_off </guard>
  <assignment> t' == t </assignment>
</transition>
```

In the component HeaterTemplate, the parameters do not have fixed values. We now define a new component, called System, based on HeaterTemplate, in which we bind the parameters to fixed values. The new component only has the variable t in its interface. Since all components need to have their formal parameters fully declared, we need to include t as a formal parameter. We instantiate the component HeaterTemplate inside System with a `bind` tag, giving the instantiation a name, Heater, that has to be unique inside the parent component:

```
<component id="System">
  <param name="t" type="real" d1="1" d2="1" local="false" dynamics="any
      " />
  <bind component="HeaterTemplate" as="Heater">
  ...
  </bind>
</component>
```

Inside the bind tag, we need to map every formal parameter in the interface of HeaterTemplate to either a formal parameter of System, or to a constant value. Here, we bind the variable t of HeaterTemplate to the variable t of Heater. The other formal parameters are bound to constants:

```
<bind component="HeaterTemplate" as="Heater">
  <map key="t">       t</map>
  <map key="r_up">    2</map>
  <map key="r_down"> -1</map>
  <map key="t_on">   18</map>
  <map key="t_off">  21</map>
</bind>
```

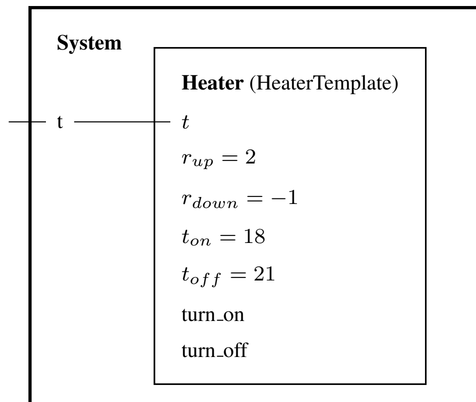


Figure 3: System component, made up of the instantiation Heater of the HeaterTemplate component

A schematic representation of the component is shown in Fig. 3.

In all, we have defined three objects:

- a component HeaterTemplate with unbound automaton-parameters,
- the instantiation Heater, in which the automaton-parameters are replaced with constant values, and
- the component System, which is made up of the instantiation Heater.

The components HeaterTemplate and System can be reused in other models. The component HeaterTemplate with its unbound parameters can be subjected to a parametric analysis in order to synthesize parameter ranges for which the automaton shows a desired behavior. All three can be used for verification using SpaceEx.

As our system consists of only a single automaton, the objects Heater and System are practically identical and seem redundant. This situation changes when we look at systems made up of more than one automaton, which is the topic of the next section.

2.2 Controller and Heater Example

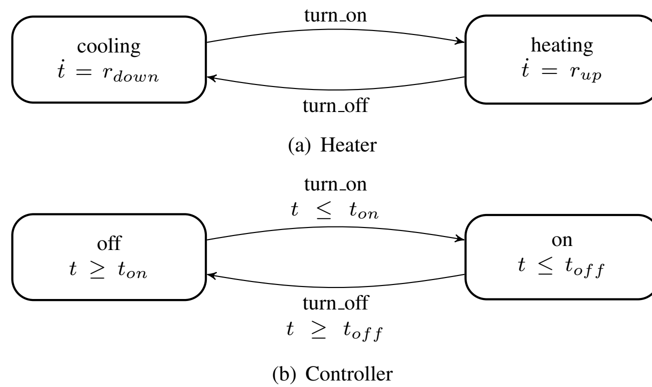
In this example we demonstrate the composition of two automata with HIOA-style parallel composition. We consider the composition of the two hybrid automata shown in Fig. 4.

Generic (template) models for a heater and a controller are declared as components. The temperature of the heater is a controlled (output) variable in the heater model and an uncontrolled (input) variable in the controller model.

Heater and controller are interacting using synchronization labels, so the labels must be declared as non-local.

The interface for the HeaterTemplate component is as follows:

```
<component id="HeaterTemplate">
  <param name="t"        type="real" d1="1" d2="1" local="false"
      dynamics="any" controlled="true" />
  <param name="r_up"     type="real" d1="1" d2="1" local="false"
      dynamics="const" />
```



(a) Heater

(b) Controller

Figure 4: Hybrid automaton network modeling a room heater

```
  <param name="r_down"   type="real" d1="1" d2="1" local="false"
      dynamics="const" />
  <param name="turn_on"  type="label" local="false"/>
  <param name="turn_off" type="label" local="false"/>
  ...
</component>
```

The interface for the HeaterTemplate component is accordingly:

```
<component id="ControllerTemplate">
  <param name="t"        type="real" d1="1" d2="1" local="false"
      dynamics="any" controlled="false" />
  <param name="t_on"     type="real" d1="1" d2="1" local="false"
      dynamics="const" />
  <param name="t_off"    type="real" d1="1" d2="1" local="false"
      dynamics="const" />
  <param name="turn_on"  type="label" local="false"/>
  <param name="turn_off" type="label" local="false"/>
  ...
</component>
```

In this model, the heater is unrestricted in the temperature range, so it has no invariant and no guards – it only determines the flow.

```
<location id="1" name="cooling">
  <flow> t' == r_down </flow>
</location>
<location id="2" name="heating">
  <flow> t' == r_up   </flow>
</location>
<transition source="1" target="2">
  <label>turn_on</label>
</transition>
<transition source="2" target="1">
  <label>turn_off</label>
</transition>
```

The controller forces transitions based on the value of the temperature, but is has no continuous variables by himself. So it has invariants and guards, but no flow or assignments.

```
<location id="1" name="off">
  <invariant> t &lt;= t_on </invariant>
```

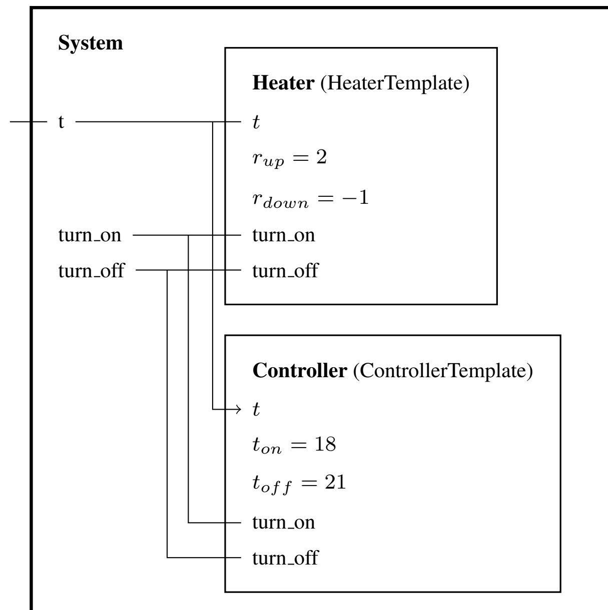


Figure 5: System component, consisting of the parallel composition of Heater and Controller

```
</location>
<location id="2" name="on">
  <invariant> t &gt;= t_off </invariant>
</location>
<transition source="1" target="2">
  <label>turn_on</label>
  <guard> t &lt;= t_on </guard>
</transition>
<transition source="2" target="1">
  <label>turn_off</label>
  <guard> t &gt;= t_off </guard>
</transition>
```

Heater and Controller are instantiated as part of the ”system” component, where the temperature variables of the two are ”connected” by binding them to the same system variable, see the schematic in Fig. 5.

We bind the temperature variable t of the heater and controller to the variable t of the System component, which is part of its interface. Similarly, the turn_on and turn_off labels are bound to the same label in System. Assuming that we don’t want any other components to interact with those labels, we declare them as local to System, so they are not part of its interface.

```
<component id="system">
  <param name="t"        type="real" d1="1" d2="1" local="false"
      dynamics="any" />
  <param name="turn_on"  type="label" local="true"/>
  <param name="turn_off" type="label" local="true"/>
  <bind component="HeaterTemplate" as="Heater">
    <map key="t">              t</map>
    <map key="r_up">           2</map>
    <map key="r_down">        -1</map>
```

```
    <map key="turn_on">   turn_on</map>
    <map key="turn_off"> turn_off</map>
  </bind>
  <bind component="ControllerTemplate" as="Controller">
    <map key="t">               t</map>
    <map key="t_on">           18</map>
    <map key="t_off">          21</map>
    <map key="turn_on">   turn_on</map>
    <map key="turn_off"> turn_off</map>
  </bind>
</component>
```

3 Restrictions

This section describes the semantic constraints that are imposed on XML files for the SX format, in addition to the constraints imposed by the RelaxNG grammar itself, which is provided in Sect. A. Many of these constraints are there to ensure that every variable in every instantiated component can be identified without ambiguity. To do so, every variable inside a component has a global name that is unique. A parser should detect violation of these constraints and report them as errors.

Notation

- A component with a least one location is called a *base component*, and called a *network component* otherwise.
- A parameter of type ”int” or ”real” is called a *variable*.
- A variable with dynamics ”const” is called a *constant*. If a constant is mapped to a numerical value inside a binding, it is called a bound constant. If a constant is mapped to a constant of its parent component, it is called an unbound constant – this is sometimes referred to as a ”parameter” of an automaton, but we will avoid this term because of its ambiguity.

Uniqueness of names

- Names of components are globally unique. There can be no two components that have the same name.
- Names of bindings are unique to a component. There can be no two bindings inside a component that have the same name.
- Names of parameters are unique to a component There can be no two parameters inside a component that have the same name.

Declaration of Parameters

- Every symbol used inside the description of a transition or location must be declared as a parameter of the parent component.
- Every non-local parameter of a component in a binding must be mapped to either a constant value or to a parameter of the parent component of the same type, dimension and dynamics.

- Local parameters can not be mapped.
- Constant values are specified as a whitespace-separated list, which is interpreted as the elements of a d1 by d2 matrix starting from the top left (1,1) and ending at the bottom right (d1,d2). The number of elements in the list must consequently be equal to d1*d2.

Example 3.1. Parameters t and r_down are declared and used:

```
<component id="HeaterTemplate">
  <param name="t"      .../>
  <param name="r_down" .../>
  <location ...>
    <flow>
      t' == r_down
    </flow>
  </location>
</component>
```

Example 3.2. In component "system", the non-local variable t of "HeaterTemplate" is mapped to the variable z. The constant r_up is mapped to the value 2. The label turn_on is not mapped since it is local.

```
<component id="HeaterTemplate">
  <param name="t"        type="real" d1="1" d2="1" local="false"
      dynamics="any" />
  <param name="r_up"     type="real" d1="1" d2="1" local="false"
      dynamics="const" />
  <param name="turn_on"  type="label" local="true"/>
</component>

<component id="system">
  <param name="z"        type="real" d1="1" d2="1" local="false"
      dynamics="any" />
  <bind component="HeaterTemplate" as="Heater">
    <map key="t">       z</map>
    <map key="r_up">    2</map>
  </bind>
</component>
```

Transitions The source and target location of a transition must be declared beforehand in the file.

Bindings The following restrictions ensure that the instantiation graph of any component is a directed acyclic graph.

- A binding can only refer to components declared beforehand in the file.
- A binding in a component can not refer to the component itself.

The following rules define how parameter properties must match and propagate in network components.

- A parameter must be mapped to a parameter with identical dimensions d1 and d2, and identical dynamics.
- A controlled parameter must be mapped to a controlled parameter.

- An uncontrolled parameter can only be mapped to a controlled parameter if at least one controlled parameter (of the same binding or another) is mapped the parameter.

Nested components With the above instantiation process, it is straightforward to create nested components. Note that recursion is not allowed.

Example 3.3. The system consists of two boilers (there is no physical meaning to the system, it's only illustration of the file format). Note that the two boilers act independently. Their labels are instantiated to different names.

```
<component id="system">
  <param name="t1" type="real" d1="1" d2="1" local="false" dynamics="
      any" />
  <param name="t2" type="real" d1="1" d2="1" local="false" dynamics="
      any" />
  <bind component="Boiler" as="Boiler1">
    <map key="t">t1</map>
  </bind>
  <bind component="Boiler" as="Boiler2">
    <map key="t">t2</map>
  </bind>
</component>
```

4 Instatiation Rules

When an SX file is parsed by the analysis tool, its base components are instantiated as hybrid automata, and its network components are instantiated as networks (parallel composition) of instantiations of its bindings (recursively). The following rules ensure that automata, networks and variables are instantiated with globally unique names.

Instantiation of components and bindings Every component and every binding is instantiated as an automaton. The automaton obtains a unique name using the following rules:

- A component defines an automaton whose name is the value of its attribute ”id”:

  <id of component>

- A binding defines an automaton whose name is the the value of the attribute ”id” of its parent component followed by ”.” followed by the value of its attribute ”as”:

  <id of parent component>.<as of binding>

Example 4.1. The following definition instantiates the automata ”HeaterTemplate”, ”system.Heater” and ”system”.

```
<component id="HeaterTemplate">
</component>

<component id="system">
  <bind component="HeaterTemplate" as="Heater">
  </bind>
</component>
```

Instantiation of parameters either a variable or label with a globally unique name, defined by the following rules:

- A parameter of a component is instantiated with the name

  $$<\text{id of component}>.<\text{name of parameter}>.$$

- The parameters of a component that is instantiated by a binding obtain a name that depends on whether they are mapped or not, which is identical to whether they are local or not. A non-mapped parameter is instantiated with the name

  $$<\text{id of parent component}>.<\text{as of binding}>.<\text{name of parameter}>.$$

  A mapped parameter is instantiated with the name of the parameter to which it is mapped.

Example 4.2. The following definition creates the variables

- HeaterTemplate.t, HeaterTemplate.r_up
- system.z

and the label HeaterTemplate.turn_on.

```
<component id="HeaterTemplate">
  <param name="t"        type="real" d1="1" d2="1" local="false"
      dynamics="any" />
  <param name="r_up"     type="real" d1="1" d2="1" local="false"
      dynamics="const" />
  <param name="turn_on"  type="label" local="true"/>
</component>

<component id="system">
  <param name="z"        type="real" d1="1" d2="1" local="false"
      dynamics="any" />
  <bind component="HeaterTemplate" as="Heater">
    <map key="t">       z</map>
    <map key="r_up">    2</map>
  </bind>
</component>
```

Explicit Dynamics If a variable has explicit dynamics, its change with time and jumps must be explicitly defined. This is to safeguard against accidental omission. As a syntactic constraint it is required that for every variable in a base component, its primed version occurs as an identifier at least once inside of every flow and assignment tag.

Example 4.3. The base component ”HeaterTemplate” has the variable ”t” with dynamics ”explicit”. Consequently, ”t’” occurs in every flow and transition

```
<component id="HeaterTemplate">
  <param name="t"        type="real" d1="1" d2="1" local="false"
      dynamics="explicit" />

  <location id="1" name="off" x="100" y="100" width="100" height="80">
    ...
    <flow> t' == r_down </flow>
  </location>
```

```
  <location id="2" name="on" x="250" y="100" width="100" height="80">
    ...
    <flow> t’ == r_up </flow>
  </location>

  <transition source="1" target="2">
    ...
    <assignment> t’ == t </assignment>
  </transition>

  <transition source="2" target="1">
    ...
    <assignment> t’ == t </assignment>
  </transition>
</component>
```

References

[1] James Clark. Jing: A relax ng validator in java. http://www.thaiopensource.com/relaxng/jing.html, 2008.

[2] James Clark. Trang: Multi-format schema converter based on relax ng. http://www.thaiopensource.com/relaxng/trang.html, 2008.

[3] James Clark, John Cowan, and Makoto Murata. Relax ng compact syntax tutorial. http://www.relaxng.org/compact-tutorial-20030326.html, 2003.

[4] Goran Frehse and Rajarshi Ray. Design principles for an extendable verification tool for hybrid systems. In *ADHS09 : 3rd IFAC Conference on Analysis and Design of Hybrid Systems*, 2009.

[5] Eric van der Vlist. *RELAX NG*. O’Reilly Media, Inc., 2003.

A RelaxNG Grammar of the SX Format

```
#
# relaxng compact grammar for xml input to sppaceex
#

# version 0.2 Release Notes
# o Changes that affect the compatibility
#  - remove element sync in transition, replace by element label
#  - changed Param+ to Param*
#  - changed Bind+ to Bind*
#  - changed Location* to Location+
#
# o Other changes
#  - 3 whitespace indentation
#  - Comments starting with double # are equivalent to an annotation
#    consisting of a documentation element from the RELAX NG DTD
#    Compatibility namespace.
#  - change version "0.1" to "0.2"
#  - added Note in param
#  - local is define even if is a label or not
#  - change "true", "false" in xsd:boolean
#  - add timedriven boolean attribute to transition
#  - add labelposition in transition, where label, guard and
    assignement go in gui
#  - add element middlepoint in transition (gui)
#  - add boolean attribute bezier in transition
#  - change Waypoints (gui)
#
default namespace = "http://www-verimag.imag.fr/xml-namespaces/sspaceex
    "

grammar {
   start =
      element sspaceex {
         attribute version { "0.2" },
         attribute math { MathFormat },

         ( Component )*
      }

   #
   ## This can either be a template automata or
   ## a network of them.  Variables and labels must be explicitly
       declared
   ## as parameters.
   #
   Component =
      element component {
         attribute id { Name },
         Note?,
         ( Param )*,
         (
            ( Location+, Transition* )
          |
            ( Bind* )
         )
      }

   #
   ## param is a thing we can abstract/have as a variable
   ## for the moment this is restricted to numbers and
```

```
## labels.  We may one day want to allow automata
## or components or sets as well.
#
Param =
   element param {
      Note?,
      attribute name { Name },

      ## local=true means non-interface variable local=false
      ## means any composition of this component in another must
          specify
      ## the value of the variable
      attribute local { xsd:boolean },
      (
         (
            attribute type { "int" | "real" },

            attribute d1 { xsd:unsignedInt | Name },

            attribute d2 { xsd:unsignedInt | Name },

            ## dynamics = const means non-evolving,
            ## any means can evolve in any way,
            ## explicit means dynamics are always specified in.
            attribute dynamics { "const" | "any" | "explicit" },

            ## variables can be controlled or uncontrolled, which
                determines
            ## how they behave in parallel composition;
            ## controlled if unspecified.
            attribute controlled { xsd:boolean }?
         )
        |
         (
            attribute type {"label"}
         )
      )
   }


#
## Like a template instantiation bind(component, x=>a1, y=>a2,...)
## we require that all the variables of the bound component are
## instantiated with values.
##
## The binding has an "as" attribute that identifies
## it by name so that local variables
## can be uniquely identified.
#
Bind =
   element bind {
      attribute component { Name },
      attribute as { Name },
      Note?,
      Map+,
      ( Position, Dim? )?
   }


#
## map is a binding for a parameter.
#
```

```
Map =
   element map {
      attribute key { Name },
      (
         list { (xsd:double+ | xsd:decimal+ ) }
       |
         Name
      )
   }


#
## location is an element of the finite state part of
## the automaton.
#
Location =
   element location {
      Note?,
      attribute id { xsd:unsignedInt },
      attribute name { Name },
      element invariant { text }?,
      element flow { text }?,

      # where it goes in the gui, and optional dimension.
      ( Position, Dim? )?
   }


#
## transition in a hybrid automaton.
#
Transition =
   element transition {
      Note?,
      ## gives id of source location.
      attribute source { xsd:unsignedInt },
      ## gives id of target location.
      attribute target { xsd:unsignedInt },

      ## whether the transition is taken as soon as enabled.
      attribute asap  { xsd:boolean }?,

      ## transition is time driven.
      attribute timedriven  { xsd:boolean }?,

      ## optional priorities.
      attribute priority { xsd:decimal }?,

      element label { Name }?,
      element guard { text }?,
      element assignment { text }?,

      element labelposition {
         Position, Dim?
      }?,

      ## position of the middle point where label, guard
      ## and assignment is attached. if not defined,
      ## the middle point take its default position.
      element middlepoint {
         Position
      }?,
```

```
         ## whether the pathway is a bezier curve, default false
         ## except in case of auto transition when bezier is true.
         attribute bezier { xsd:boolean }?,

         ## way points for graphical ui.
         Waypoints?
      }

   #
   ## waypoints give the points into a (spline)
   ## interpolation for shaping arrows between
   ## locations in the gui.
   #
   Waypoints =
      element waypoints {

         ## list of the points between source and middlepoint
         ## event number of floating point number: (x, y)+.
         element beforemiddle {
            list { (xsd:double, xsd:double)+ }
         }?,

         ## list of the points between middlepoint and target
         ## event number of floating point number: (x, y)+.
         element aftermiddle {
            list { (xsd:double, xsd:double)+ }
         }?
      }


   Position =
      (
         attribute x { xsd:double },
         attribute y { xsd:double }
      )


   Dim =
      (
         attribute width { xsd:double },
         attribute height { xsd:double }
      )


   Note =
      element note { text }


   Name =
      xsd:string { pattern = "[a-zA-Z_][a-zA-Z0-9_]*" }

   MathFormat = "SpaceEx"
}
```

B SX File of the Heater Example

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!--
```

```
This is a basic example to demonstrate the SpaceEx file format for a
    single
automaton.
A generic (template) heater model with parameters (temperature rates
    and thresholds)
is declared as a component.
It is instantiated as part of the "system" component, and each of the
    parameters
is attributed a value.
-->

<sspaceex xmlns="http://www-verimag.imag.fr/xml-namespaces/sspaceex"
          version="0.2"
          math="SpaceEx">
  <component id="HeaterTemplate">
      <note>
        This is a simple model of a heater. The temperature t rises
        with rate r_up if the heater is on, and falls
        with rate r_down if the heater is off.
        The heater switches on and off at thresholds t_off and
        t_on.
        It can be verified that the heater keeps the temperature
        between t_off and t_on.
      </note>
    <param name="t"        type="real" d1="1" d2="1" local="false"
        dynamics="any" />
    <param name="r_up"     type="real" d1="1" d2="1" local="false"
        dynamics="const" />
    <param name="r_down"   type="real" d1="1" d2="1" local="false"
        dynamics="const" />
    <param name="t_on"     type="real" d1="1" d2="1" local="false"
        dynamics="const" />
    <param name="t_off"    type="real" d1="1" d2="1" local="false"
        dynamics="const" />
      <param name="turn_on"  type="label" local="true"/>
      <param name="turn_off" type="label" local="true"/>

      <location id="1" name="off" x="100" y="100" width="100" height="
          80">
        <note>In this location the temperature is falling because the
            heater is off. </note>
        <invariant>
          t &lt;= t_on
        </invariant>
      <flow>
        t’ == r_down
      </flow>
    </location>

    <location id="2" name="on" x="250" y="100" width="100" height="80">
        <note>In this location the temperature is rising because the
            heater is on. </note>
        <invariant>
          t &gt;= t_off
        </invariant>
      <flow>
        t’ == r_up
      </flow>
    </location>

    <transition source="1" target="2">
```

```
      <note>When the temperature reaches the lower threshold, it
          switches on.</note>
      <label>turn_on</label>
      <guard>
        t &lt;= t_on
      </guard>
      <assignment>
        t' == t
      </assignment>
      </transition>

    <transition source="2" target="1">
      <note>When the temperature reaches the upper threshold, it
          switches off.</note>
      <label>turn_off</label>
      <guard>
        t &gt;= t_off
      </guard>
      <assignment>
        t' == t
      </assignment>
      </transition>
  </component>

  <component id="system">
      <note>
        The system consists of a single heater.
        The temperatures of the heater is part of the system interface.
        Since the synchronization labels are local to the heater model,
        and are not part of the system interface.
      </note>
    <param name="t"        type="real" d1="1" d2="1" local="false"
        dynamics="any" />
    <bind component="HeaterTemplate" as="Heater">
      <map key="t">t</map>
      <map key="r_up">2</map>
      <map key="r_down">-1</map>
      <map key="t_on">18</map>
      <map key="t_off">21</map>
    </bind>
  </component>
</sspaceex>
```

C SX File of the Controller-Heater Example

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!--
This is a basic example to demonstrate the SpaceEx file format for the
    composition of
two automata with HIOA parallel composition.
Generic (template) models for a heater and a controller are declared as
     components.
The temperature of the heater is a controlled (output) variable in the
    heater model
and an uncontrolled (input) variable in the controller model.
The two are instantiated as part of the "system" component, where the
    temperature
variables of the two are "connected" by binding them to the same system
     variable.
```

```
Heater and controller are interacting using synchronization labels, so
    the labels
must be declared as non-local.

In this model, the heater is unrestricted in the temperature range, so
    it has no
invariant and no guards, it only determines the flow.
The controller forces transitions based on the value of the temperature
    , but is
has no continuous variables by himself. So it has invariants and guards
    , but no
flow or assignments.
-->

<sspaceex xmlns="http://www-verimag.imag.fr/xml-namespaces/sspaceex"
    version="0.2" math="SpaceEx">
  <component id="HeaterTemplate">
    <note>This is a simple model of a heater.
The temperature t rises with rate r_up if the heater is on, and falls
    with rate r_down if the heater is off.
The heater switches on and off by synchronizing with the controller on
    the labels turn_on and turn_off.
The temperature t is a controlled (output) variable of the heater.</
    note>
    <param name="t" type="real" d1="1" d2="1" local="false" dynamics="
        any" controlled="true" />
    <param name="r_up" type="real" d1="1" d2="1" local="false" dynamics
        ="const" />
    <param name="r_down" type="real" d1="1" d2="1" local="false"
        dynamics="const" />
    <param name="turn_on" type="label" local="false" />
    <param name="turn_off" type="label" local="false" />
    <location id="1" name="off" x="67.0" y="189.0" width="100" height="
        80">
      <note>In this location the temperature is falling because the
          heater is off.</note>
      <flow>t' == r_down</flow>
    </location>
    <location id="2" name="on" x="303.0" y="189.0" width="100" height
        ="80">
      <note>In this location the temperature is rising because the
          heater is on.</note>
      <flow>t' == r_up</flow>
    </location>
    <transition source="1" target="2">
      <note>Switching the heater on.</note>
      <label>turn_on</label>
      <assignment>t' == t</assignment>
      <labelposition x="0.0" y="-56.0" />
      <middlepoint x="183.0" y="102.0" />
    </transition>
    <transition source="2" target="1">
      <note>Switching the heater off.</note>
      <label>turn_off</label>
      <assignment>t' == t</assignment>
      <labelposition x="0.0" y="10.0" />
      <middlepoint x="185.0" y="284.0" />
    </transition>
  </component>
  <component id="ControllerTemplate">
    <note>The controller switches the heater on and off at thresholds
        t_off and t_on.
```

```
It can be verified that the controller keeps the temperature between
    t_off and t_on.
The controller switches the heater on and off by synchronizing with on
    the labels turn_on and turn_off.
The temperature t is an uncontrolled (input) variable of the controller
    .</note>
    <param name="t" type="real" d1="1" d2="1" local="false" dynamics="
        any" controlled="false" />
    <param name="t_on" type="real" d1="1" d2="1" local="false" dynamics
        ="const" />
    <param name="t_off" type="real" d1="1" d2="1" local="false"
        dynamics="const" />
    <param name="turn_on" type="label" local="false" />
    <param name="turn_off" type="label" local="false" />
    <location id="1" name="off" x="67.0" y="171.0" width="100" height="
        80">
      <note>Waiting for the temperature to rise.</note>
      <invariant>t &lt;= t_on</invariant>
    </location>
    <location id="2" name="on" x="308.0" y="170.0" width="100" height="
        80">
      <note>Waiting for the temperature to fall.</note>
      <invariant>t &gt;= t_off</invariant>
    </location>
    <transition source="1" target="2">
      <note>When the temperature reaches the lower threshold, switch
          the heater on.</note>
      <label>turn_on</label>
      <guard>t &lt;= t_on</guard>
      <labelposition x="0.0" y="-47.0" />
      <middlepoint x="177.5" y="101.0" />
    </transition>
    <transition source="2" target="1">
      <note>When the temperature reaches the upper threshold, switch
          the heater off.</note>
      <label>turn_off</label>
      <guard>t &gt;= t_off</guard>
      <labelposition x="0.0" y="10.0" />
      <middlepoint x="181.5" y="249.0" />
    </transition>
  </component>
  <component id="system">
    <note>The system consists of a heater and a controller.
The temperatures of the heater is part of the system interface.

The temperature variables of the two are "connected" by binding them to
     the same system variable.

Heater and controller are interacting using synchronization labels, so
    the labels must be declared as non-local.</note>
    <param name="t" type="real" d1="1" d2="1" local="false" dynamics="
        any" />
    <param name="turn_on" type="label" local="true" />
    <param name="turn_off" type="label" local="true" />
    <bind component="HeaterTemplate" as="Heater" x="180.0" y="184.0">
      <map key="t">t</map>
      <map key="r_up">2</map>
      <map key="r_down">-1</map>
      <map key="turn_on">turn_on</map>
      <map key="turn_off">turn_off</map>
    </bind>
```

```
    <bind component="ControllerTemplate" as="Controller" x="179.0" y="
        29.0">
      <map key="t">t</map>
      <map key="t_on">18</map>
      <map key="t_off">21</map>
      <map key="turn_on">turn_on</map>
      <map key="turn_off">turn_off</map>
    </bind>
  </component>
</sspaceex>
```

D SX File of the Two-Boiler Example

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!--
This example shows how components can be nested.
There is no physical meaning to the system, it's only illustration
of the file format.

Note that the two boilers act independently.
Their labels are instantiated to different names.
-->

<sspaceex xmlns="http://www-verimag.imag.fr/xml-namespaces/sspaceex"
          version="0.2"
          math="SpaceEx">
  <component id="HeaterTemplate">
      <note>
        This is a simple model of a heater. The temperature t rises
        with rate r_up if the heater is on, and falls
        with rate r_down if the heater is off.
        The heater switches on and off by synchronizing
        with the controller on the labels turn_on and turn_off.
        The temperature t is a controlled (output) variable of the
        heater.
      </note>
    <param name="t"        type="real" d1="1" d2="1" local="false"
        dynamics="any" controlled="true" />
    <param name="r_up"     type="real" d1="1" d2="1" local="false"
        dynamics="const" />
    <param name="r_down"   type="real" d1="1" d2="1" local="false"
        dynamics="const" />
      <param name="turn_on"  type="label" local="false"/>
      <param name="turn_off" type="label" local="false"/>

      <location id="1" name="off" x="100" y="100" width="100" height
          ="80">
        <note>In this location the temperature is falling because the
            heater is off. </note>
      <flow>
        t' == r_down
      </flow>
    </location>

    <location id="2" name="on" x="250" y="100" width="100" height="80">
        <note>In this location the temperature is rising because the
            heater is on. </note>
      <flow>
        t' == r_up
      </flow>
```

```
  </location>

  <transition source="1" target="2">
    <note>Switching the heater on.</note>
    <label>turn_on</label>
    <assignment>
      t' == t
    </assignment>
    </transition>

  <transition source="2" target="1">
    <note>Switching the heater off.</note>
    <label>turn_off</label>
    <assignment>
      t' == t
    </assignment>
    </transition>
</component>

<component id="ControllerTemplate">
    <note>
      The controller switches the heater on and off at thresholds
          t_off and
      t_on. It can be verified that the controller keeps the
          temperature
      between t_off and t_on.
      The controller switches the heater on and off by synchronizing
      with on the labels turn_on and turn_off.
      The temperature t is an uncontrolled (input) variable of the
      controller.
    </note>
  <param name="t"        type="real" d1="1" d2="1" local="false"
      dynamics="any" controlled="false" />
  <param name="t_on"     type="real" d1="1" d2="1" local="false"
      dynamics="const" />
  <param name="t_off"    type="real" d1="1" d2="1" local="false"
      dynamics="const" />
    <param name="turn_on"  type="label" local="false"/>
    <param name="turn_off" type="label" local="false"/>

    <location id="1" name="off" x="100" y="100" width="100" height
        ="80">
      <note>Waiting for the temperature to rise. </note>
      <invariant>
        t &lt;= t_on
      </invariant>
  </location>

  <location id="2" name="on" x="250" y="100" width="100" height="80">
      <note>Waiting for the temperature to fall. </note>
      <invariant>
        t &gt;= t_off
      </invariant>
  </location>

  <transition source="1" target="2">
    <note>When the temperature reaches the lower threshold, switch
        the heater on.</note>
    <label>turn_on</label>
    <guard>
      t &lt;= t_on
    </guard>
```

```
      </transition>

    <transition source="2" target="1">
      <note>When the temperature reaches the upper threshold, switch
          the heater off.</note>
      <label>turn_off</label>
      <guard>
        t &gt;= t_off
      </guard>
      </transition>
  </component>

  <component id="Boiler">
      <note>
        A boiler consists of a heater and a controller.
        The temperatures of the heater is part of the interface.

        The temperature variables of the two are "connected" by
        binding them to the same system variable.

      Heater and controller are interacting using synchronization
          labels, so the labels
      must be declared as non-local.
      </note>
    <param name="t"        type="real" d1="1" d2="1" local="false"
        dynamics="any" />
      <param name="turn_on"  type="label" local="true"/>
      <param name="turn_off" type="label" local="true"/>
    <bind component="HeaterTemplate" as="Heater">
      <map key="t">t</map>
      <map key="r_up">            2</map>
      <map key="r_down">         -1</map>
      <map key="turn_on">turn_on</map>
      <map key="turn_off">turn_off</map>
    </bind>
    <bind component="ControllerTemplate" as="Controller">
      <map key="t">t</map>
      <map key="t_on">           18</map>
      <map key="t_off">          21</map>
      <map key="turn_on">turn_on</map>
      <map key="turn_off">turn_off</map>
    </bind>
  </component>

  <component id="system">
      <note>
        The system consists of two boilers.
        The two act independently.
      </note>
    <param name="t1"        type="real" d1="1" d2="1" local="false"
        dynamics="any" />
    <param name="t2"        type="real" d1="1" d2="1" local="false"
        dynamics="any" />
    <bind component="Boiler" as="Boiler1">
      <map key="t">t1</map>
    </bind>
    <bind component="Boiler" as="Boiler2">
      <map key="t">t2</map>
    </bind>
  </component>
</sspaceex>
```