Modeling Techniques, Programming Languages, and Design Toolsets for Hybrid Systems

Luca P. Carloni, Maria Domenica DiBenedetto, Roberto Passerone, Alessandro Pinto and Alberto Sangiovanni-Vincentelli
Abstract

This report is a critical review of existing modeling techniques, programming languages, and design toolsets for hybrid systems. We analyze industrial and academic tools with the intent of comparing their applicability and the different models used to support the tools. In addition to the review, we also provide comments and recommendations on how to build a standard interchange format and a standard representation language for hybrid systems that will enable better interaction between groups working on the design of embedded controllers based on hybrid system technology.
## Modelica

4.1 The `MODELICA` Language Syntax ................. 40
4.2 The `MODELICA` Language Semantics ................. 44
4.3 Examples ........................................ 46
  4.3.1 The bouncing-ball example ......................... 49
  4.3.2 Equivalent Mathematical Description of a `MODELICA` Program ......................... 51
4.4 Discussion .................................... 52

## Sildex

5.1 The `SILDEX` Approach .......................... 54
5.2 Synchronous Paradigm and Synchronous Languages ..... 56
5.3 The `SILDEX` Integrated Toolset .................. 58
5.4 Discussion .................................... 58

## HyVisual

6.1 `HYVISUAL` Graphical Syntax ..................... 59
6.2 `HYVISUAL` Semantics ............................. 61
6.3 Examples ....................................... 63
6.4 Discussion ..................................... 65

## Scicos

7.1 Signals and Blocks .............................. 68
7.2 Basic Blocks .................................. 71
  7.2.1 Continuous Basic Blocks (CBB) .................. 71
  7.2.2 Discrete Basic Blocks (DBB) ................... 73
  7.2.3 Zero Crossing Basic Blocks (ZCBB) ............. 74
  7.2.4 Synchro Basic Blocks (SBB) .................... 74
  7.2.5 Synchronization .............................. 75
7.3 Interface from `SCICOS` to `SYNDEX` ............... 76
7.4 Example ....................................... 79
7.5 Discussion ..................................... 79
### III Tools for Formal Verification

#### 8 Charon

8.1 The Charon Language .................................................. 82
   8.1.1 Architectural Hierarchy ......................................... 82
   8.1.2 Behavioral Hierarchy ............................................. 82
8.2 The Syntax ................................................................. 84
8.3 The Semantics ............................................................... 86
8.4 Compositionality Results .............................................. 87
8.5 The Charon Toolkit ....................................................... 88
8.6 Examples ........................................................................ 88

#### 9 CheckMate

9.1 Main Features ................................................................. 97
9.2 The CheckMate Model ...................................................... 99
9.3 Building a CheckMate Model ........................................... 100
9.4 Formal Verification in CheckMate ..................................... 102
9.5 Discussion ...................................................................... 104

#### 10 Masaccio

10.1 The Masaccio Language .................................................. 105
10.2 The Syntax ..................................................................... 106
10.3 The Semantics ............................................................... 107
10.4 Component Operators ..................................................... 109
10.5 Assume-Guarantee in Masaccio ...................................... 110

#### 11 Shift

11.1 The Shift language: syntax and semantics at a glance ...... 113
   11.1.1 The world ............................................................. 113
   11.1.2 Types ................................................................... 114
   11.1.3 Time and components ........................................... 115
   11.1.4 The world automaton ............................................ 116
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.2</td>
<td>Discussion</td>
<td>117</td>
</tr>
<tr>
<td>12</td>
<td>Hysdel</td>
<td>119</td>
</tr>
<tr>
<td>12.1</td>
<td>HYSDEL Syntax</td>
<td>119</td>
</tr>
<tr>
<td>12.2</td>
<td>HYSDEL Semantics</td>
<td>122</td>
</tr>
<tr>
<td>12.3</td>
<td>Examples</td>
<td>123</td>
</tr>
<tr>
<td>12.4</td>
<td>Discussion</td>
<td>129</td>
</tr>
<tr>
<td>IV</td>
<td>Semantics</td>
<td>130</td>
</tr>
<tr>
<td>13</td>
<td>The LSV Tagged-Signal Model</td>
<td>131</td>
</tr>
<tr>
<td>13.1</td>
<td>The LSV Model</td>
<td>132</td>
</tr>
<tr>
<td>13.1.1</td>
<td>Processes</td>
<td>133</td>
</tr>
<tr>
<td>13.1.2</td>
<td>Concurrency and Communication</td>
<td>134</td>
</tr>
<tr>
<td>13.2</td>
<td>Basic Time</td>
<td>135</td>
</tr>
<tr>
<td>13.3</td>
<td>Treatment of Time in Processes</td>
<td>136</td>
</tr>
<tr>
<td>13.4</td>
<td>Implementing Concurrency and Communication</td>
<td>136</td>
</tr>
<tr>
<td>13.5</td>
<td>Representing Models of Computation with the LSV Model</td>
<td>137</td>
</tr>
<tr>
<td>13.6</td>
<td>Discussion</td>
<td>138</td>
</tr>
<tr>
<td>14</td>
<td>Trace Algebras</td>
<td>139</td>
</tr>
<tr>
<td>14.1</td>
<td>Introduction</td>
<td>140</td>
</tr>
<tr>
<td>14.2</td>
<td>The Trace Algebra Approach</td>
<td>141</td>
</tr>
<tr>
<td>14.2.1</td>
<td>Traces and Trace Algebra Operators</td>
<td>143</td>
</tr>
<tr>
<td>14.2.2</td>
<td>Homomorphisms and Conservative Approximations</td>
<td>145</td>
</tr>
<tr>
<td>14.3</td>
<td>Discussion</td>
<td>147</td>
</tr>
<tr>
<td>15</td>
<td>Behavioral Approach</td>
<td>149</td>
</tr>
<tr>
<td>15.1</td>
<td>The Behavioral Approach</td>
<td>150</td>
</tr>
<tr>
<td>15.1.1</td>
<td>Input/Output Representation</td>
<td>151</td>
</tr>
<tr>
<td>15.1.2</td>
<td>Modular Representation of Behavioral Systems</td>
<td>152</td>
</tr>
<tr>
<td>15.1.3</td>
<td>Controllability and Observability</td>
<td>154</td>
</tr>
</tbody>
</table>
15.2 Hybrid Systems and the Behavioral Approach .......................... 155
  15.2.1 Behavioral State Representation of Hybrid Systems ............... 155
  15.2.2 Concatenability of Behaviors in Hybrid Systems .................... 158
  15.2.3 Robust Hybrid Control Using the Behavioral Approach ............... 159
15.3 Discussion: Behavioral Systems and Trace Algebras ..................... 162

V Potential Interchange Formats ................................................. 165

16 Hybrid System Interchange Format ........................................... 166
  16.1 Introduction ................................................................. 166
  16.2 Syntax ...................................................................... 167
    16.2.1 Hybrid automaton .................................................. 167
    16.2.2 Network of hybrid automata ..................................... 169
  16.3 Semantics ................................................................. 171
    16.3.1 Semantics of a single automaton ................................ 171
    16.3.2 Semantics of the network ....................................... 173
  16.4 Communication in the Network ....................................... 176
  16.5 Discussion ............................................................... 181

17 Metropolis ................................................................. 184
  17.1 The METROPOLIS Methodology ....................................... 185
  17.2 The METROPOLIS Meta-Model ....................................... 187
    17.2.1 Behaviors as actions ............................................ 188
    17.2.2 Action automata .................................................. 189
    17.2.3 Processes and media ............................................. 191
    17.2.4 Refinement ......................................................... 196
    17.2.5 Coordination constraints ...................................... 199
    17.2.6 Quantity constraints ............................................ 200
  17.3 Discussion ............................................................... 201
### 18 Proposal for a new interchange format 202

18.1 Features to be included in the language 202

18.2 Language syntax 205

18.3 Language semantics 207

18.4 Examples 210

18.4.1 Cut-off Control 214

### 19 Concluding Remarks 223

19.1 Comparative Summary 223

19.2 Towards the Development of a Standard Interchange Format 227

### Bibliography 230
Part I

Foundations
Chapter 1

Introduction

Technology advances allow us to design systems whose complexity was simply unthinkable a few years ago. Design time has become the bottleneck for bringing new products to market. Traditional design paradigms are no longer effective. The most challenging designs are in the area of safety critical systems such as the ones used to control the behavior of transportation systems (e.g., airplanes, cars, and trains) or industrial plants. The difficulties reside in accommodating constraints on functionality and implementation. Functionality has to guarantee correct behavior under diverse states of the environment and potential failures, implementation has to meet cost, size, and power consumption constraints.

When designing embedded systems of this kind, it is essential to take into consideration all effects including the interaction between environment (plant to be controlled) and design (digital controller). This calls for methods that can deal with heterogeneous components that exhibit a variety of different behaviors. For example, digital controllers can be represented mathematically as discrete event systems, while plants are mostly represented by continuous time systems whose behavior is captured by partial or ordinary differential equations. In addition, the complexity of the plants is such that representing them at the detailed level is often impractical or
even impossible. To cope with this complexity, abstraction is a very powerful method. Abstraction consists in eliminating details that do not affect the behavior of the system that we may be interested with. In both cases, different mathematical representations have to be mixed to analyze the overall behavior of the controlled system.

Many are the difficulties in mixing different mathematical domains. *In primis*, the very meaning of interaction may be challenged. In fact, when heterogeneous systems are interfaced, interface variables are defined in different mathematical domains that may be incompatible. This aspect makes verification and synthesis impossible, unless a careful analysis of the interaction semantics is carried out.

In general, pragmatic solutions precede rigorous approaches to the solution of engineering problems. This case is no exception. Academic institutions and private software companies (e.g. Mathworks) started developing computational tools for the simulation, analysis, and implementation of control systems deploying first common sense reasoning and then trying a formalization of the basic principles. These approaches focused on a particular class of heterogeneous systems: systems featuring the combination of discrete-event and continuous-time subsystem. Recently, these systems have been the subject of intense research by the academic community because of the interesting theoretical problems arising from analysis and design of these systems as well as of the relevance in practical applications. These systems are called *hybrid systems*.

**Simulink**, **Stateflow** and **Matlab** together provide excellent modeling and simulation capability for the design capture and the functional verification via simulation of embedded systems; however, often there is a need to subject the models (developed in Simulink) to a more rigorous and domain-specific analysis as well as to refine this high-level description into an implementation. In addition, we expect that no single design framework will be capable of encompassing all the needs of system designers. Hence,
exporting and importing design representations will be a necessity even for future powerful tools. Remodeling the system in another tool’s modeling language while possible requires substantial manual effort. Additionally, maintaining consistency between models is error-prone and difficult in the absence of tool support. The popularity of MATLAB, SIMULINK, and STATEFLOW implies that significant efforts have already been invested in creating a large model-base in SIMULINK/STATEFLOW. It is desirable that application developers take advantage of this effort without foregoing the capabilities of their own analysis and synthesis tools. Owing to these factors a strong need has been expressed for automated semantic translators that can interface with and translate the SIMULINK/STATEFLOW models into the models of different analysis and synthesis tools.

On a more fundamental level, a unified approach to hybrid systems modeling is needed to enable the use of joint techniques and a formal comparison between different approaches and solutions. Suggesting the guidelines to use for the development of a common interchange language for hybrid systems modeling is the main objective of the WPHS work-package in Columbus.

The first step in this direction has been collecting data on available languages, formalism and tools that have been proposed in the past years. In particular, we focus on hybrid-system languages that are the basis for some popular industrial verification tools such as SIMULINK and STATEFLOW, SILDEx, and for some academic work, CHECKMate, CHARON, MASACCIO, SCICOS-SYNDEEx, HyVISUAL, HYSDel, Modelica, and Metropolis. We will stress that the roots of the difficulties in combining tools and environments for hybrid system design lay in the semantics aspects.

For the semantics issues, our focus is on general approaches that could be used to provide the back bone we are looking for. The most general angle is taken by using denotational models where traces play a major role. The first instance of the use of denotational approaches was to the best of our recollection, the work by Willems on the behavioral approach to system theory.
The work by Lee and Sangiovanni-Vincentelli on the tagged-signal model is the first attempt at providing a general framework for comparing different models of computation. This approach has been used as the theoretical framework on communication protocols and some aspects of combining synchronous and asynchronous deployments (both software and hardware) of distributed networks. The work on trace algebra originated by the work of Burch in his thesis at Stanford has provided a more powerful and general environment where guidelines can be given to guarantee that properties are met when combining heterogeneous models of computation. The behavioral approach of Willems can be cast into this framework showing its generality. While more general, the trace algebraic approach is complex and relies upon notions of abstract algebras that are not widely known.

As per the recommendation for a standard interchange format, there are a number of possible solutions. All of them require some degree of modifications or extensions to present models and tools. While we believed at the onset of the project that a final recommendation was indeed possible, the results of our analysis is that it is premature to dictate the characteristics of a commonly agreed standard. We do analyze the approaches that seem to us technically correct and feasible and in the final Chapter we outline some solutions. However, we need to reach out for the entire hybrid system community to gain wide acceptance for the proposal. To do so, Networks of Excellence such as HyCon and Artist seem the appropriate avenues to propagate our views. We discuss the present situation and the lack of a complete solution throughout this report.

This report that encompasses DHS1, 2, and 3 is organized in parts and chapters. Chapter 2 is about the formal mathematical definition of hybrid systems.

Part II discusses the most relevant tools for simulation and design of hybrid and embedded systems. With respect to the industrial offering, we discuss the Simulink/Stateflow design environment in Chapter 3 the
<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Web Page</th>
<th>References</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charon</td>
<td>Univ. of Pennsylvania</td>
<td><a href="http://www.cis.upenn.edu/mobies/charon/">www.cis.upenn.edu/mobies/charon/</a></td>
<td>[3, 4, 8]</td>
<td>8</td>
</tr>
<tr>
<td>CheckMate</td>
<td>Carnegie Mellon Univ.</td>
<td><a href="http://www.ece.cmu.edu/~webb/checkmate/">www.ece.cmu.edu/~webb/checkmate/</a></td>
<td>[122]</td>
<td>9</td>
</tr>
<tr>
<td>Dymola</td>
<td>Dynasim AB</td>
<td><a href="http://www.dynamia.se/">www.dynamia.se/</a></td>
<td>[56]</td>
<td>4</td>
</tr>
<tr>
<td>HSIF</td>
<td>Vanderbilt University</td>
<td><a href="http://micc.isis.vanderbilt.edu:8080/HSIF">http://micc.isis.vanderbilt.edu:8080/HSIF</a></td>
<td>[72, 109]</td>
<td>16</td>
</tr>
<tr>
<td>Masaccio</td>
<td>UC Berkeley</td>
<td>ytcarnot.eecs.berkeley.edu/hyvisual</td>
<td>[65]</td>
<td>6</td>
</tr>
<tr>
<td>Metropolis</td>
<td>UC Berkeley</td>
<td><a href="http://www.eecs.berkeley.edu/~lab">www.eecs.berkeley.edu/~lab</a></td>
<td>[79]</td>
<td>10</td>
</tr>
<tr>
<td>Modelica</td>
<td>Modelica Association</td>
<td><a href="http://www.modelica.org">www.modelica.org</a></td>
<td>[60, 130, 59]</td>
<td>4</td>
</tr>
<tr>
<td>Shift</td>
<td>UC Berkeley</td>
<td><a href="http://www.path.berkeley.edu/shift">www.path.berkeley.edu/shift</a></td>
<td>[47, 48]</td>
<td>11</td>
</tr>
<tr>
<td>SynDEx</td>
<td>INRIA</td>
<td>www-synq.inria.fr/syndex</td>
<td>[68, 69]</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 1.1: References for the various modeling approaches, toolsets.

MODELICA language and the modeling and simulation tool DYMOLA based on it in Chapter 4, and Sildeox in Chapter 5. Among the academic tools, we summarize the key features of Scicos and SYNDEX in Chapter 7, HyViSual in Chapter 6, and Shift in Chapter 11.

Part III focuses on tools for formal verification of hybrid systems. In particular, Chapter 8 discusses Charon, Chapter 9 discusses CheckMate, and Chapter 10 discusses Masaccio.

Table 1.1 lists the various tools and modeling approaches that we discuss in this report together with the corresponding chapters. It reports also the main institution that supports the development of each project as well as pointers to the corresponding web site and some relevant publications.

Part IV focuses on semantics. In particular, Chapter 13 presents the Lee and Sangiovanni-Vincentelli (LSV) tagged-signal model, a denotational model for comparing models of computation. Chapter 14 introduces trace algebras as a novel approach to modeling concurrency in heterogeneous systems using trace theory. Chapter 15 describes the behavioral approach to system theory and describes the relationship between this model and the trace algebra approach.

Part V discusses two potential interchange formats. First, Chapter 16...
Introduction

summarizes the Hybrid System Interchange Format proposed by Vanderbilt. Then, Chapter 17 focuses on the METROPOLIS meta-model as a potential unifying framework for tools and models.

Finally, Chapter 19 presents future work and some concluding remarks.
Chapter 2

Preamble

In this chapter, we give a formal definition of a general hybrid system as us in the control community. Most models used in the control community can be thought of as a special case of this general model. We will then discuss some general issues about features that formal frameworks and methods should have. We will use these criteria as axes in our discussion about existing approaches.

2.1 Formal Definition of Hybrid Systems

The notion of a hybrid system that has been used in the control community is centered around a particular composition of discrete and continuous dynamics. In particular, the system has a continuous evolution and occasional jumps. The jumps correspond to the change of state in an automaton whose transitions are caused either by controllable or uncontrollable external events or by the continuous evolution. A continuous evolution is associated to each state by means of ordinary differential equations. The structure of the equations and the initial condition may be different for each state. While this informal description seems rather simple, the precise definition of the evolution of the system is quite complex.
Early work on formal models for hybrid systems includes phase transition systems [2] and hybrid automata [102]. These somewhat simple models were further generalized with the introduction of compositionality of parallel hybrid components in hybrid I/O automata [101] and hybrid modules [11]. In the sequel, we follow the classic work of Lygeros et al. [100] to formally describe a hybrid system as used in the control literature. We believe that this model is sufficiently general to form the basis of our work in the future chapters.

**Definition 1 (Hybrid Systems)** A continuous time (resp. discrete time) hybrid system is a tuple $\mathcal{H} = (Q, U_D, X, U, V, S_C, S, E, Inv, R, G)$, (resp. $\mathcal{H} = (Q, U_D, X, U, V, S_D, S, E, Inv, R, G)$) where:

- $Q = \{q_i, i \in J\}$, $J = \{1 \ldots N\}$ is the set of discrete states;
- $U_D = U_{D_{\text{EXT}}} \cup U_{D_{\text{CONTR}}}$ is the set of discrete inputs;
- $X, U, V$ are subsets of finite dimensional vector spaces and are respectively the continuous state, input and disturbance space. We denote by $U_C$ the class of measurable control functions $u : T \rightarrow U$ and by $U_d$ the class of measurable disturbance functions $\delta : T \rightarrow V$, where $T$ denotes the set of reals $\mathbb{R}$ (resp. the set of integers $\mathbb{Z}$).
- $S_C$ is a subclass of continuous time dynamical systems (resp. $S_D$ is a subclass of discrete-time dynamical systems).
  - $S_i \in S_C$ is defined by the equation:
    $$\dot{x}(t) = f_i(x(t), u(t), \delta(t)) \quad i \in J$$
    where $t \in \mathbb{R}$, $x(t) \in X$ and $f_i$ is a function such that, $\forall u \in U_C$, $\forall \delta \in U_d$, the solution $x(t)$ exists and is unique.
  - $S_h \in S_D$ is defined by the equation:
    $$x(t + 1) = f_i(x(t), u(t), \delta(t)) \quad i \in J$$
    where $t \in \mathbb{Z}$, $x(t) \in X$ and $f_i$ is a vector field;
2.1 Formal Definition of Hybrid Systems

- \( S : Q \rightarrow S_C \) (resp. \( S : Q \rightarrow S_D \)) is a mapping associating to each discrete state a continuous time (resp. a discrete time) dynamical system;

- \( E \subset Q \times U_D \times Q \) is a collection of discrete transitions;

- \( Inv : Q \rightarrow 2^{X \times U_D \times U \times V} \) is a mapping called invariant;

- \( R : E \times X \times U \times V \rightarrow 2^X \) is the reset mapping;

- \( G : E \rightarrow 2^{X \times U \times V} \) is a mapping called guard.

The triple \((Q, U_D, E)\) can be viewed as a Finite State Machine (FSM) having state set \(Q\), inputs \(U_D\) and transitions defined by \(E\). This FSM characterizes the structure of the discrete transitions.

Discrete transitions, as defined by the set \(E\), can be of different types:

- if \(\sigma \in U_{D_{EXT}}\), the transition is forced by a discrete disturbance and is called a switching transition;

- if \(\sigma \in U_{D_{CONTR}}\), the transition is determined by a controllable input event and is called a controllable transition.

- if an invariance condition is not satisfied, a transition called an invariance transition occurs.

Switching and invariance transitions are both uncontrolled transitions.

Let \(\mathbb{N}\) be the set of non negative integers. Following \[100\], we introduce the concept of hybrid time basis for the temporal evolution of the system.

**Definition 2** (Hybrid Time Basis) A hybrid time basis \(\tau\) is a finite or an infinite sequence of sets \(I_j, j \in \{0, \ldots, L\}, \ L \in \mathbb{N}\) such that:

- \(I_j = \{t \in T : t_j \leq t \leq t_j', j < L\}; \) and if \(L\) is finite, \(I_L = \{t \in T : t_L \leq t < t_L'\}\) or \(I_j = \{t \in T : t_L \leq t \leq t_L'\};\)
• For all $j$, $t_j \leq t'_j = t_{j+1}$.

Let $T$ be the set of all hybrid time bases. We define now an execution of a hybrid system that describes its state evolution in time.

**Definition 3 (Hybrid System Execution)** An execution $\chi$ of a hybrid system $\mathcal{H}$, with initial state $\left( \hat{q}, x_0 \right) \in Q \times X$, is a collection $\chi = (\hat{q}, x_0, \tau, \sigma, q, u, \delta, \xi)$ with $\tau \in T$, $\sigma : \tau \rightarrow U_D$, $q : \tau \rightarrow Q$, $u \in U_C$, $\delta \in U_d$, $\xi : T \times \mathbb{N} \rightarrow X$ satisfying:

1. **Discrete evolution:**
   
   \[
   q(I_0) = \hat{q}; \quad q(I_{j+1}) : e_j \in E, \quad e_j = (q(I_j), \sigma(I_{j+1}), q(I_{j+1}))
   \]

2. **Continuous evolution:** the function $\xi$ satisfies the conditions

   - $\xi(t_0, 0) = x_0$
   - $\xi(t_{j+1}, j + 1) \in R\left(e_j, \xi(t'_j, j), u(t'_j), v(t'_j)\right)$
   - $\xi(t, j) = x(t) \quad \forall t \in I_j$

   where $x(t)$ is the solution at time $t$ of the dynamical system $S_h = S(q(I_j))$, with initial condition $x(t_j) = \xi(t_j, j)$, given some control input function $u \in U_C$ and some disturbance function $\delta \in U_d$.

   - if $t_j < t'_j$, then $\left(\xi(t, j), \sigma(I_j), u(t), v(t)\right) \in Inv(q(I_j)) \quad \forall t \in [t_j, t'_j]$
   - if $\tau$ is a finite sequence and $t'_j \neq t'_{j+1}$, then $\left(\xi(t'_j, j), u(t'_j), v(t'_j)\right) \in G(e_j)$

If a discrete disturbance and a discrete control act simultaneously and an invariance condition is no longer satisfied, the hybrid execution of a hybrid system $\mathcal{H}$ is not well defined according to the definition above. In this case, we could take two avenues: one is to allow non determinism in the choice
of reaction of the system, taking into account all possible cases. A second, is to establish priorities among transitions. In particular, in this report, we assume that the switching transition has the highest priority and the invariance transition has higher priority than the controllable transition. Then $\mathcal{H}$ is well defined in all cases. Note that simultaneous events are the cause of most of the inconsistencies and ambiguities in models for hybrid systems.

**Definition 4** A hybrid system execution is said to be (i) trivial if $\tau = \{I_0\}$ and $t_0 = t_0'$; (ii) finite if $\tau$ is a finite sequence ending in a right closed interval; (iii) infinite if $\tau$ is an infinite sequence or $\sum_{j=0}^{\text{card}(\tau)-1} t_j' - t_j = \infty$; (iv) Zeno, if $\tau$ is infinite but $\sum_{j=0}^{\infty} t_j' - t_j < \infty$.

Beside the presence of simultaneous input events, other sources of non determinism are embedded in the definition of the discrete system, so that, given a particular input at a state, the next state map associates to this input a set of possible transitions. In this case, a correct analysis of the behavior of the system should be able to reproduce all possible executions, clearly a very expensive proposition. Some of the languages used in verification systems require the elimination of any non deterministic behavior or they simply apply heuristic rules to select one of the next states.

### 2.2 Syntax and Semantics

In the terminology of formal languages \(^1\), the terms syntax and semantics are used to classify and categorize distinct aspects of language characteristics.

The **syntax** of a language describes the structure and composition of allowable phrases and sentences of the language. However syntax itself is devoid of meaning: it simply establishes what strings are valid and how they may be parsed or decomposed. The meaning of these syntactic elements must be provided through **semantics**.

---

\(^1\) and also of natural languages, of course
Consider, for example, the syntax and semantics of an assignment statement. As a syntactic construct, an assignment statement might consist of a variable and an expression (themselves syntactic constructs), separated by the token = as an assignment operator. Semantically, the variable denotes a location in computer memory, while the expression denotes computation of a value based on the contents of memory. Overall, the semantics of assignment is to perform the expression evaluation based on current memory contents and then update the value stored in the particular location corresponding to the variable.

When it comes to modeling hybrid systems, semantics can be classified according to two distinct perspectives; thus we have:

- operational semantics;
- denotational semantics.

Operational semantics describes systems according to their intrinsic computational capabilities. It therefore gives an implicit characterization of the system behavior. For example, the operational semantics of a continuous-time dynamical system is represented by the set of differential equations describing it along with the related initial conditions. More generally, the operational approach to describe systems semantics consists of a methodology that assign values in a specified domain to the syntactic constructs modeling the system.

A useful and most used modeling paradigm for describing semantics of a system from an operational point of view is that of transition systems. There is no standard definition for such a class of systems (for a survey, see for example [80]), roughly speaking however they consist of:

1. a set of states $S$;
2. a transition relation $\mathcal{R} \subset S \times S$ on the set of states.
2.3 Abstraction and Hierarchy

A state can be seen as a collection of values in a specified domain containing all the necessary information for describing the future evolution of the system (given a transition relation). The transition relation, instead, can be thought of as the law describing all the possible sequences of states.

The class of transition systems is rich enough to include many other modeling paradigms, like Finite State Machines for purely discrete systems, differential equations for continuous-time dynamical systems and also hybrid systems (see [110]).

Denotational semantics, on the other hand, describe systems in terms of their visible behavior. A single behavior of the system is called a trace. For a purely discrete un-timed system, for example, a trace is a sequence of symbols in a given alphabet, while for continuous-time dynamical systems, a trace is the collection of all input and output signals.

Unlike the operational approach, the denotational semantic description of a given system is simply a collection of traces. In this perspective any transition relation encoding the system behavior is unnecessary since we already have a complete semantic description of the systems through the enumeration of all its possible traces (see [98] for more details).

2.3 Abstraction and Hierarchy

Hybrid systems are the composition of discrete event and continuous time dynamics behaviors. They are clearly best suited to model embedded systems acting with an analogue environment, without any compromise between having either a detailed description or a meaningful formal model.

However, this interaction between discrete and continuous components makes the analysis (and design, of course) of hybrid systems an extremely complex issue. Such a complexity is even augmented if one considers hybrid systems consisting of many distinct components, for example in applications like coordinating robot systems [5], automotive control [20], aircraft
control [131], chemical process control systems [57], automated highway systems [18, 46, 45].

It is therefore necessary to analyze the behavior of complex hybrid systems, to have suitable formal tools to manage in some way the intrinsic complexity of such systems. The most important are:

- abstraction
- hierarchy

Even if possessing quite different meanings, these two conceptual methodologies are strictly intertwined and even complementary.

Abstraction refers to the practice of isolating, in a given system, only the properties of interest, while suppressing all irrelevant details. All modern object-oriented software design techniques support abstraction, hence making such an issue quite well developed and structured. In particular, as far as hybrid systems modeling is concerned, the major aspects that are usually implemented by hybrid systems modeling languages are:

- encapsulation;
- reuse.

Encapsulation provides abstraction by information hiding. In other words, given a system sub-component, only signals by means of which interaction with other components is made possible will be visible externally. Reuse or instantiation refers to the possibility of grouping all components exhibiting an identical behavior into the same class of objects. The properties of individual objects can thus easily be derived from those of the basic object class.

Hierarchy, on the other hand, allows one to study the properties of a complex system by:

1. analyzing system sub-components independently, and
2.4 Compositionality

2. infer properties of the overall system from those of the sub-components and from the way they interact with each other.

Since, in their most used form, hybrid systems have the structure of a finite-state machine at the discrete layer of abstraction, it would be useful, if not necessary, to apply the hierarchy paradigm to these mathematical objects. Hierarchical state machines have been first introduced in State-charts [76] and are nowadays included in the majority of software design and modeling paradigms, most notably UML [31], which is becoming a standard in the software specification community.

The notion of hierarchy is also present in reactive modules [7] and tools like Stateflow (http://www.mathworks.com), Ptolemy [44], and Hy-Chart [70].

2.4 Compositionality

2.4.1 Compositional Semantics

In a compositional setting, the behavior of a given system must be inferred from the behavior of its sub-components. This is perhaps the most important consequence arising from an appropriate hierarchical modeling of systems.

From a formal semantics point of view, the equivalence of two systems can be determined by the observation of their executions (i.e. traces). Compositional semantics means that the semantics of a component can be determined by the semantics of its sub-components. Such a property is characteristic, for example, of Communicating Sequential Processes (CSP) [82] or of Calculus of Communicating Systems (CCS) [105] and is desirable -if not necessary- for the model checking and systematic design of systems with a high degree of complexity.

Importing such a compositional semantics in the hybrid systems framework is more difficult, most notably because we have to introduce the notion
of global time shared among different hybrid components.

Thus, a desirable feature of a hybrid systems modeling language would be that of supporting compositional semantics at all levels of abstraction; this can be obtained by reconstructing the semantics of a hybrid system by the observation of its traces (observational trace semantics).

The compositionality of semantics, then, implies that the behavior of a system can be easily determined by the observation of traces of its sub-components.

2.4.2 Compositional Refinement

If a language supports a compositional semantics, the methodology of compositional refinement can be applied to verify if a given hybrid system satisfy specification constraints.

Simply stated, the concept of compositional refinement is as follows: assume that we have a specification design $S$ and that we obtain an implementation design $I$ from $S$ by locally replacing some sub-component $N$ in $S$ by a subcomponent $M$. Then, the implementation $I$ refines the specification $S$ iff sub-system $M$ refines sub-system $N$.

2.5 Assume-Guarantee Reasoning

One of the major cornerstones that compositional semantics analysis relies on is the assume-guarantee principle. We have already said that the compositional verification of hybrid systems properties is based on a modular decomposition of a specified system into its sub-components, on which the properties of interest are verified independently. However, there is usually the need to put a given component into the right context, that is we have to consider the interaction of the component with its external environment (i.e. other components).

The assume-guarantee principle gives a precise formalization to such an
2.5 Assume-Guarantee Reasoning

issue: assuming for simplicity a systems built out of two components (clearly a component can be composite, i.e. include other sub-modules) $A$ and $B$, properties of component $A$ are analyzed assuming that $B$ behaves correctly (with respect to some design specification), and subsequently $B$ is verified assuming the correctness of $A$.

The assume-guarantee methodology therefore gives a natural framework for the implementation of such a circular reasoning paradigm and represents an extremely useful tool for the formal verification of complex systems. Hybrid modeling languages supporting assume-guarantee principle clearly have a desirable feature which can be exploited for facilitating the analysis of correctness of particularly complex systems.

Modeling paradigms supporting assume-guarantee for embedded systems are defined in [106, 1, 104, 12] for systems resulting from the parallel composition of reactive components, and in [7] for systems supporting serial compositions. In hybrid systems modeling it is however necessary to describe complex behaviors by means of nested parallel and serial composition. The only modeling language supporting assume-guarantee reasoning for arbitrarily composed hybrid systems is, at the moment being, MASACCIO (see Chapter 10).

In [22], Benveniste describes a behavioral framework for the specification of hybrid systems which facilitates compositionality. In turns, such specification formalism results in programs that are not in operational form (i.e., they cannot be directly executed), thus making both timing and causality analysis necessary. Benveniste discusses also how to automatically synthesize proper scheduling constraints for the joint simulation of discrete-time, event-based, and continuous-time components.
Part II

Tools for Simulation and Design
Chapter 3

Simulink and Stateflow

In this chapter we describe the data models of SIMULINK and STATEFLOW. The information provided below is derived from the SIMULINK documentation as well as by “reverse engineering” SIMULINK/STATEFLOW models.

SIMULINK and STATEFLOW are two interactive tools that seamlessly integrate with MATLAB, the popular environment for technical computing that is based on the high-performance MATLAB language and is marketed by The MathWorks. MATLAB integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. SIMULINK is an interactive tool for modeling and simulating nonlinear dynamic systems. It can work with linear, nonlinear, continuous-time, discrete-time, multi-variable, and multirate systems. STATEFLOW is an interactive design and development tool for complex control and supervisory logic problems. STATEFLOW supports visual modeling and simulation of complex reactive systems by simultaneously using finite state machine (FSM) concepts, STATECHARTS formalisms, and flow diagram notations. A STATEFLOW model can be included in a

\footnote{We have also drawn from a draft report (Simulink and Stateflow Data Model) by Sandeep Neema, Institute for Software Integrated Systems Vanderbilt University, Nashville.}
Simulink and Stateflow

Simulink model as a subsystem.

Together with Simulink and Stateflow, Matlab has become the de facto design-capture standard in academia and industry for control and data-flow applications that mix continuous and discrete-time domains. The graphical input language and the powerful simulation and symbolic manipulation tools available in the tool-set ease the task of system designers. The tools are based on a particular mathematical formalism, language, necessary to analyze and simulate the design. Unfortunately, the semantics of the language are not formally defined. As we discuss below, the behavior of the design depends upon the execution of the associated simulation engine and the engine itself has somewhat ambiguous executions rules.

3.1 Simulink / Stateflow Syntax

Both Simulink and Stateflow are graphical languages. Simulink graphical syntax is very intuitive (and this is also the reason why this language is very popular). Like Scicos (see Chapter 7) and HyVisual (see Chapter 6), Simulink has a rich library of components that can be used to compose a system. There are six fundamental block-sets (libraries) for modeling systems (Figure 3.1):

- **Continuous**: this library contains blocks for processing continuous signals like Derivative and Integrator blocks. There are also more complex continuous time operators like the State-Space blocks that can be used to model dynamical systems described by state equations. Other general blocks are Zero-Pole blocks to describe transfer functions in the s domain.

- **Discrete**: this library contains blocks for processing discrete signals. Most of the blocks are description of transfer functions in the z domain. Discrete Zero-Pole, Discrete State-Space, and Discrete-Time Integrator are examples of blocks that can be instantiated and pa-
3.1 Simulink / Stateflow Syntax

rameterized in a Simulink model. This library also includes a Zero-Order Hold and a First-Order Hold.

- **Math Operations**: this is a general library of blocks representing mathematical operations like Sum, Dot Product, and Abs (absolute value).

- **Sinks**: many types of sink blocks are provided in this library. There are several types of display for run time graph generation. It possible to store simulation results in a MATLAB workspace variable for post-processing. Output ports are a special type of Sinks blocks from this library, and can be used to define subsystems that will be used as a elementary blocks in other systems.

- **Sources**: this library contains signal generators of different nature that can be used as stimuli for test-benches. Input ports are a special type of SOURCES.

- **Discontinuities**: this library contains non linear transformations of signals. Saturation and Quantizer are examples of such transformations. The Hit Crossing block is very useful for modeling hybrid systems. This block has a threshold parameter. Hit Crossing generates an output event when the threshold is hit.

The Simulink syntax supports the definition of subsystems that can be instanced in a Simulink model. A subsystem is just like any other model. This feature allows users to describe a hierarchical model.

A special type of block that can be instantiated in a Simulink model is a Stateflow model. The syntax of Stateflow is similar to that of Statecharts. A Stateflow model is a set of states connected by arcs. A state is represented by a rounded rectangle. A state can be refined in a Stateflow diagram. A Stateflow model can have data input/output ports as well as event input/output ports. Both inputs and outputs can
Figure 3.1: A snapshot of library blocks
3.2 The *Simulink* Data Model

be defined as internal or external, i.e. coming from the *Simulink* parent model.

Each arc has a label with the following syntax:

```
event[condition]{condition_action}/transition_action
```

The transition is taken when the event is enabled and the condition is true (both condition and event can be missing). If the transition is enabled, the condition action is executed. The transition action is executed when the transition to the next state is taken.

A state has a label with the following syntax:

```
name/
entry:entry action
during:during action
exit:exit action
on event_name:on event_name action
```

*tt* name denotes the name of the state; the *entry* action is executed when the state is entered; the *during* action is executed whenever the model is evaluated and the state cannot be left; the *exit* action is executed when the state is left; finally, the *event_name* action is executed each time the specified event is enabled.

3.2 The *Simulink* Data Model

*Simulink* is a simulation environment that supports the analysis of mixed discrete-time and continuous-time models. Different simulation techniques are used according to whether continuous-time subsystems and/or discrete-time subsystems are present. Since our focus is on hybrid systems, we discuss only on the case in which both components are present.

A *Simulink* project \(^2\) is stored in an ASCII text file in a specific format referred to as Model File Format in the *Simulink* documentation. The

\(^2\)In order to avoid any ambiguity, a complete model of a system in *Simulink* will be referred to as a *Simulink* project.
Simulink project files are suffixed with “.mdl” and therefore we may occasionally refer to a Simulink project file as an mdl file. There is a clear decoupling between the Simulink and the Stateflow models. When a Simulink project contains Stateflow models, the Stateflow models are stored in a separate section in the mdl file. We discuss Stateflow models separately in the next section. It must be remarked that the data models presented here capture only the information that is being exposed by Simulink in the mdl file. Note that a substantial amount of semantic information that is sometimes required for the effective understanding of the Simulink models is hidden in the MATLAB simulation engine, or in Simulink primitive library database.

The Simulink simulation engine deals with the components of the design using the continuous-time semantic domain as a unifying domain whenever both continuous and discrete-time components are present. The simulation engine includes a set of integration algorithms, called solvers, which are based on the MATLAB ordinary differential equation (ODE) suite.

A sophisticated ODE solver uses a variable time-step algorithm that adaptively selects a time-step tuned to the fastest mode of the system. The algorithm allows for errors in estimating the correct time-step and backtracks whenever the truncation error exceeds a bound given by the user. The algorithm is conservative. All signals of the system have to be evaluated at the time-step dictated by the integration algorithm even if no event is present at these times. A number of multi-rate integration algorithms have been proposed for ODEs to improve the efficiency of the simulators but they have a serious overhead that may make them even slower than the original conservative algorithm.

The most difficult part for a mixed-mode simulator that has to deal with discrete-events as well as continuous time dynamics is to manage the interaction between the two domains. In fact, the evolution of the continuous-time dynamic may trigger a discrete-event. The trigger may be controlled by the
3.2 The Simulink Data Model

value of a continuous variable, in which case detecting when the variable assumes a particular value is of great importance. The time in which the value is crossed is essential to have a correct simulation. This time is often difficult to obtain accurately. In particular, simulation engines have to use a sort of bisection algorithm to bracket the time value of interest. Numerical noise can cause serious accuracy problems. SIMULINK has a predefined block called zero-crossing that forces the simulator to accurately detect the time a particular variable assumes the zero value.

In SIMULINK, there is the option of using fixed time-step integration methods. The control part of the simulator simplifies considerably, but there are a few problems that may arise. If the system is stiff, i.e., there are substantially different time constants, the integration method has to use a time step that, for stability reason, is determined by the fastest mode. This yields an obvious inefficiency when the fast modes die out and the behavior of the system is determined only by the slower modes. In addition, an a priori knowledge of the time constants is needed to select the appropriate time step. Finally, not being able to control the time step may cause the simulation to be inaccurate in estimating the time at which a jump occurs or even miss the jump altogether!

Variable computations are scheduled according to the time step. Whenever there is a static dependency among variables at a time step, a set of simultaneous algebraic equations have to be solved. In sophisticated solvers, Newton-like algorithms are used to compute the solution of the set of simultaneous equations. When the design is an aggregation of subsystems, subsystems may be connected in ways that yield a degree of ambiguity in the computation. For example, assume that subsystem $A$ has two outputs, one goes to subsystem $B$ and one to subsystem $C$. Subsystem $B$ has an output that feeds $C$. In this case, we may evaluate the output of $C$ whenever we have computed one of its inputs. Assuming that $A$ has been processed, then we have the choice of evaluating the outputs of $B$ or of $C$. Depending on
the choice of processing $B$ or $C$, the outputs of $C$ may have different values! Simultaneous events may in fact yield a nondeterministic behavior. In fact, both cases are in principle correct behaviors unless we load the presence of connections among blocks with causality semantics. In this case, $B$ has to be processed before $C$. Like many other simulators, SIMULINK deals with non determinism with scheduling choices that cannot be but arbitrary if a careful and often times expensive causality analysis is not carried out. Even when a causality analysis is present, there are cases where the non determinism cannot be avoided since it is intrinsic in the model. In this case, scheduling has to be somewhat arbitrary. If the user knew what scheme is used and had some control on it, he/she may adopt the scheduling algorithm that better reflects what he/she had in mind. However, if the choice of the processing order is done inside the simulator according, for example, to a lexicographical order, changing the name of the variables or of the subsystems may change the behavior of the system itself! Since the inner workings of the simulation engines are often not documented, strange results and inconsistencies may occur. This phenomenon is well known in hardware design when Register Transfer Languages (RTL) are used to represent a design and an RTL simulator is used to analyze the system. For example, two different RTL simulators may give two different results even if the representation of the design is identical, or if it differs solely on the names of the subsystems and on the order in which the subsystems are entered.

### 3.3 The Stateflow Data Model

STATEFLOW models the behavior of dynamical systems based on finite state machines (FSM) concepts. The STATEFLOW modeling formalism is derived from STATECHARTS developed by Harel [76], and the STATEFLOW models follow the same semantics. The essential differences from STATECHARTS are in the action language. The STATEFLOW action language has been extended primarily to reference MATLAB functions, and MATLAB workspace variables.
3.3 The Stateflow Data Model

Additionally, a concept of condition action has been added to the transition expression. A condition action is performed every time the condition is evaluated.

The interaction between Simulink and Stateflow occurs at the event and data boundaries. The simulation of a system consisting of Simulink and Stateflow models is carried out by releasing the control of the execution alternatively to the two simulation engines embedded in the two tools. In the hardware literature, this mechanism is referred to as co-simulation. Since control changes from one engine to the other, there is an overhead that may be quite significant when events are exchanged frequently. An alternative simulation mechanism would be to build a unified engine. This, however, would require a substantial overhaul of the tools and of the underlying semantic models. We believe this should be done in a not too distant future.
3.4 Examples

In this section we use the bouncing ball example to illustrate some important modeling aspects of Simulink/Stateflow. A first naïve model of such a simple system is shown in Figure 3.2. The Simulink model has an instance of a Stateflow diagram. Variable $g$, representing the gravity, is integrated to get the ball velocity which, in turn, is integrated to get the ball position. The velocity integrator (indicated with $v$ in the diagram) has two extra inputs: a reset and an initial condition. This integrator is controlled by the Stateflow block which decides when to reset the initial condition.

The Stateflow model has two states (even if only one would be sufficient): down is the initial mode representing a state of the system in which the ball is falling down. up indicates that the ball is going up. The initial state is indicated by the entry point in the Stateflow diagram and sets the initial velocity to zero. When the ball hits the ground the state changes.
3.4 Examples

Figure 3.4: Simulation result of the bouncing ball example: A) without hit-crossing detector, B) using hit-crossing detector.

to up and the integrator is reset to a new initial condition.

Unfortunately simulation results of such a naive model are very poor as illustrated in Figure 3.4. The reason is that, at each time step, the SIMULINK simulator activates the STATEFLOW block, which executes and sees if a transition has to be taken. In this model, threshold-crossing detection is done in the STATEFLOW block and there is no way for the simulator to change the simulation step in order to exactly locate the event.

A better model is shown in Figure 3.3. Here, we have introduced a hit-crossing block from the SIMULINK Discontinuities library, which contains blocks whose outputs are discontinuous functions of their inputs. With this new block, the SIMULINK simulator is able to locate exactly the time when the position variable hits zero. In other words, we have instrumented the simulator with a mechanism that enables the use of a variable step solver to simulate the systems. A comparison of the simulation results for both
cases is given in Figure 3.4.

3.5 Discussion

The MATLAB toolbox with SIMULINK and STATEFLOW provides excellent modeling and simulation capabilities for control and data-flow applications mixing continuous and discrete-time domains.

However, often there is a need to subject the models (developed in SIMULINK) to a more complex, rigorous, and domain-specific analysis. In fact, we have seen that the behavior of the system is sensitive to the inner working of the simulation engines. Consequently, fully understanding what takes place inside the tools would be important to prevent unpleasant surprises. However, in most cases, users ignore these details and may end up with an erroneous result without realizing it. Indeed, the lack of formal semantics of the models used inside this very successful tool set has been considered a serious drawback in academic circles\(^3\) thus motivating an intense activity in formalizing the semantics of hybrid systems and a flurry of activities aimed at providing translation to and from SIMULINK/STATEFLOW. Remodeling the system in the analysis tool’s modeling language, while possible, requires substantial manual effort. Additionally, maintaining consistency between the SIMULINK/STATEFLOW models and the analysis tool’s models is error-prone and difficult in the absence of tool support. Also, the popularity of MATLAB, SIMULINK, and STATEFLOW implies that significant effort has already been invested in creating a large model-base in SIMULINK/STATEFLOW. It is desirable that application developers take advantage of this effort without foregoing the capabilities of their own analysis and synthesis tools. Owing to these factors, a strong need has been expressed for automatic semantic translators that can interface with and

\(^3\)Notice that some authors dispute the fact that “SIMULINK has not semantics” by arguing instead that SIMULINK has a multitude semantics (depending on user-configurable options) which, however, are informally and sometimes partially documented [38].
3.5 Discussion

translate the Simulink/Stateflow models into the models of different analysis and synthesis tools. In [38] Caspi et al. discuss a method for translating a discrete-time subset of Simulink models into Lustre programs. The proposed method consists of three steps (type inference, clock inference, and hierarchical bottom-up translation) and has been implemented in a prototype tool called S2L.

---

4While doing so, they also attempt to formalize the typing and timing mechanisms of such discrete-time subset of Simulink.
Chapter 4

Modelica

MODELICA is a language for hierarchical physical modeling [130, 59]. Modelica 1.0 was announced in September 1997 and the current version is Modelica 2.0. It is an object oriented language for mathematical modeling targeting efficient simulation. One of its most important features is non-causal modeling, meaning that users do not need to directly specify input and output signals but rather they define variables and equations that must be satisfied. Modelica provides a formal type system for this modeling effort. Two commercial modeling and simulation environments for Modelica are currently available: DYMOLA [56] (Dynamic Modeling Laboratory) is a modeling and simulation tool based on the Modelica language and marketed by Dynasim AB. MathCore Engineering markets MathModelica, a Modelica simulation environment which is closely integrated into Mathematica and Microsoft Visio.

4.1 The Modelica Language Syntax

The Modelica syntax is described in [15]. Readers familiar with object oriented programming will find some similarities with Java and C++ but there are also fundamental differences since the Modelica language is ori-
4.1 The Modelica Language Syntax

Modelica is a typed language. It provides some primitive types like Integer, String, Boolean and Real. As in C++ and Java, it is possible to build more complicated data types by defining classes. There are many types of classes: records, types, connectors, models, blocks, packages and functions. The meaning of these classes will be explained below.

Classes, as well as models, have fields (variables they act on) and methods. C++ or Java programmers are used to this terminology where methods are functions that are part of a class definition. In Modelica, class methods are represented by equation and algorithms sections. An equation is syntactically defined as expression = expression and an equation section may contain a set of equations. The syntax supports non-causal modeling or, in other words, the possibility to describe a model as a set of equations on variables, instead of as a method of computing output values by operating on input values. In non-causal modeling there is no distinction between input and output variables; rather, variables are involved in equations that must be satisfied. The Algorithm sections are simply sequential blocks of statements and are closer to Java or C++ programming from a syntactic and semantic viewpoints.

Modelica also allows the users to specify causal models by defining functions. A function is a special class that can have inputs, outputs, and an algorithm section which specifies the model behavior.

Before going into the details of variables declaration, it is important to introduce the notion of variability of variables. A variable can be continuous-
time, discrete-time, a parameter or a constant depending on the modifier used in its instantiation. The MODELICA variability modifiers are discrete, parameter and constant. Even if the meaning is auto-explanatory, the formal semantics is given in Section 4.2.

MODELICA also defines a connect operator that takes two variable references as parameters. Connections are like other equations. In fact, connect statements are translated into particular equations that involve the required variables. Variables must be of the same type (either continuous-time or discrete-time). connect statements are a convenient shortcut for the user that could write his/her own set of equations to relate variables that are “connected”.

MODELICA is a typed system. Users of the language can extend the predefined type set by defining new, and more complex, types. The MODELICA syntax supports the following classes 1:

- **record**: Records are just an aggregations of types without any method definition. In particular, no equations are allowed in the definition or in any of its components, and they may not be used in connections. A record is a heterogeneous set of typed fields.

- **type**: may only be extension to the predefined types, records, or array of type. It is like a typedef in C++.

- **connector**: A connector is a special type for variables that will be involved in a connection equation. Connectors are specifically used to connect models. No equations are allowed in their definition or in any of their components.

- **model**: A model describes the behavior of a physical system by means of equations. It may not be used in connections.

---

1Some of the constructs mentioned below are explained in 4.2
4.1 The Modelica Language Syntax

- **block**: A block describes an input-output relation. It has fixed causality. Each component of an interface must either have causality equal to input or output. It can not be used in connections.

- **package**: may only contain declarations of classes and constants.

- **function**: has the same restrictions as for blocks. Additional restrictions are: no equations, at most one algorithm section. Calling a function requires either an algorithm section or an external function interface which is a way of invoking a function described in a different language (for instance C). A function can not contain calls to the Modelica built-in operators der, initial, terminal, sample, pre, edge, change, reinit, delay, and cardinality whose meaning is explained in Section 4.2.

Inheritance is allowed through the keyword extends like in Java. A class can extend another class thus inheriting its parent class fields, equations, and algorithms. A class can be defined to be partial, meaning that it cannot be instantiated directly but it has to be extended first.

The Modelica language provides control statements and loops. There are two basic control statements (if and when) and two loop statements (while and for).

- **if expression then**
  
  equation/algorith

  else

  equation/algorith

  end if

For instance, an expression can check the values of a continuous variable. Depending on the result of the boolean expression, a different set of equations is chosen. It is not possible to mix equations and algorithms. If one branch has a model described by equations, so has to have the other branch. Also the number of equations has to match.
• for IDENT in expression loop
  { equation/algorithm ; }
end for

IDENT is a valid MODELICA identifier. A for loop can be used to generate a vector of equations, for instance. It is not possible to mix equations and algorithms.

• when expression then
  { equation/algorithm ; }
end when

• when expression then
  { equation/algorithm ; }
else when expression then
  { equation/algorithm ; }
end when

Real variables assigned in a when clause must be discrete-time. Also equations in a when clause must be of the form $v = expression$.

Expressions use relation operators like $\leq, \geq, ==, ...$ on continuous time variables, but can be any other valid expression whose result is a boolean.

### 4.2 The MODELICA Language Semantics

This section explains the meaning of a MODELICA program starting from its basic building blocks.

MODELICA distinguishes between discrete-time and continuous time variables. Continuous-time variables are the only ones that can have a non-zero derivative. MODELICA has a predefined operator $\text{der}(v)$ which indicates the time derivative of the continuous variable $v$. When $v$ is a discrete time variable (specified by using the discrete modifier at instantiation time) the derivative operator should not be used even if we can informally say that its
4.2 The Modelica Language Semantics

derivative is always zero and changes only at event instants (for a definition of event instant see Section 4.3.2). Parameter and constant variables remain constant during transient analysis.

The second distinction to point out is between the algorithm and the equation sections. Both are used to describe the behavior of a model. An equation section contains a set of equations that must be satisfied. Equations are all concurrent and the order in which they are written is immaterial. Furthermore, an equation does not distinguish between input and output variables. For instance, an equation could be $i_1(t) + i_2(t) = 0$ which does not specify if $i_1$ is used to compute $i_2$ or vice-versa. The value of $i_1$ and $i_2$, at a specific time $t_0$, is set in such a way that all the equations of our model (and hence also the previous one) are satisfied.

An algorithm section is a block of sequential statements. Here the order matters. In an algorithm section, the user should use the assignment operator := instead of the equality operator =. Only one variable reference can be used as left operand. The semantics is that the value of the variable to the left is computed using the values of the variables to the right of the assignment operator.

Causal models in Modelica are described using functions. A function is a particular class that has input and output variables. A function has exactly one algorithm section which specifies the input/output behavior of the function. Non-causal models are described by means of equation sections defined in classes or models.

Statements like if then else and for are quite intuitive. In the case of if clauses in equation sections, if the switching conditions contains also variables that are not constants or parameters then the else branch cannot be omitted otherwise the behavior will not be defined when a false expression is evaluated.

When clauses deserve a particular attention. When the switching expression (see Section 4.1) evaluates to true the body of the when clause is
considered to be part of the model and, in particular, all its equations must be satisfied. If the switching expression evaluates to false, all the output variables are held to their last values. Hence, it is important that if the when clause is in an equation section, every equality operator has only one component instance on the left-hand side (otherwise it is not clear which variable should be held). Such component instance will be the one whose value is held while the switching expression evaluates to false. This condition can be checked by a syntax checker.

Finally, we discuss the connect statement, which is just another way of expressing certain equations. A connect statement can generate two kinds of equations depending on the nature of the variables that are passed as arguments. If the variables are declared to be flows (using the flow modifier at instantiation time) then the connection of variables $v_1, \ldots, v_n$ generates the equation $v_1 + \ldots + v_n = 0$. If the variables are not flows then their connection generates the equation $v_1 = \ldots = v_n$.

### 4.3 Examples

To better understand how Modelica is used for specifying physical systems, consider the example shown in Figure 4.1. The circuit has two states:

![Circuit Diagram](image)

Figure 4.1: Architectural diagram of the Modelica example
4.3 Examples

charging and discharging. In the charging state, the value of DC is 5V while in the discharging state its values is −5V. We first define all the components and types of our circuit. First of all, the quantities we are interested in are voltages and currents that can be defined as types in MODELICA [60]:

```modelica
type Voltage = Real;
type Current = Real;
```

Now we can define a generic pin of a component to be characterized by a voltage and a current:

```modelica
connector Pin
  Voltage v;
  flow Current i;
end Pin;
```

There are two keywords here: connector denotes that this class is used in connections. And flow which means that the sum of all Current field of Pins in a connection must be equal to zero. A generic two-pin component can be described in the following way [60]:

```modelica
partial class TwoPin
  Pin p, n;
  Voltage v;
  Current i;
  equation
    v = p.v - n.v;
    0 = p.i + n.i;
    i = p.i;
end TwoPin;
```

This defines a positive and a negative pin. Kirchoff’s equations for voltage and current are declared in the equation section. This class is partial and we extend it to specify two pins components like resistors and capacitors. A capacitor for instance can be described as follows:
class Capacitor
    extends TwoPin;
    parameter Real C(unit="F") "Capacitance";
    equation
        C*der(v) = i;
end Capacitor;

The equation section has only to declare the component constituent equation since the other equations are inherited from a two-pins component. A parameter is used for the capacitance value.

class circuit
    Resistor R1(R=10);
    Capacitor C1(C=0.01);
    Vsource DC;
    Ground G;
    equation
        connect (DC.p, R1.p);
        connect (DC.n, G.p);
        connect (C1.n, DC.n);
        connect (R1.n, C1.p);
    algorithm
        when C1.p.v > 4 then
            DC.VA := -5;
        elsewhen C1.p.v < 1 then
            DC.VA := 5;
        end when;
end circuit;

The circuit illustrated in 4.1 is described using an equation section and an algorithm section. The equation section has a set of connect statements that, together with the equations defined in the components, complete the expression of the Kirchoff’s laws. The algorithm section is described using
4.3 Examples

an imperative language based on assignments :=. Depending on the voltage value on the capacitor C1, the value of the voltage source is computed by the algorithm. Figure 4.2 reports the waveforms obtained by simulating this circuit with the DYMOLA simulator.

Figure 4.2: Simulation results for the circuit example

4.3.1 The bouncing-ball example

In this section the bouncing ball example is modeled and simulated using the DYMOLA simulator. The following equations characterize such system:

\[ \ddot{x} = -g ; \quad x(0) = 1 ; \quad \dot{x}(0) = 0 \]  \hspace{1cm} (4.1)

Also there is a change in \( \dot{x} \) when the ball touches the ground. In particular, each time it touches the ground the speed changes direction loosing also some energy. This can be modeled as an hybrid system, where the ball speed is reset each time it hits the ground. The corresponding MODELICA code is very simple:

```modelica
class bball
    Real x;
    Real y;
```

Real xd;
parameter Real s = 0.8;
equation
  der(xd) = -9.8;
  der(y) = 1;
  der(x) = xd;
algorithm
  if initial() then
    reinit(x,1);
    reinit(y,0);
    reinit(xd,0);
  end if;
  when x <= 0 then
    reinit(xd,-s*xd);
  end when;
end bball;

The model has a parameter \( s \) to describe the energy lost when the ball touches the ground. The `initial()` keyword returns true if the simulation time is equal to the start time (in our case zero). We use the `reinit` method that takes a continuous variable (differentiated) and an expression and resets the value of the variable to the value of the expression at a specific event instant (in this particular case when the ball hits the ground). Figure 4.3 shows the simulation results. A bouncing ball is a typical hybrid system that manifests a Zeno behavior. Figure 4.3 clearly shows that at some point the value of \( x \) will constantly decrease assuming negative value. The equations describing the model are still satisfied. The reason why this happens is that eventually the value of \( x \) becomes negative due to numerical errors. For \( x < 0 \) the model has more than one solution: one that reverses the ball speed and another that continues decreasing \( x \).
4.3 Examples

A non-expert MODELICA user, could be tempted to use an if statement rather than a when statement. The meaning of the model would be very different. An if statement has an expression which is “stateless” meaning that it is tested regardless of its previous value. Expression in a when statement evaluates to true when its value changes from false to true. It is indeed possible to convert a when statement to an if statement using the keyword edge. An edge(expression) is true if the expression was false in the previous integration step and is true in the current integration step. Statements like when expression then can be converted to if edge(expression) then.

4.3.2 Equivalent Mathematical Description of a MODELICA Program

A MODELICA program can be interpreted by defining a one to one mapping between the program and a system of Differential Algebraic Equations (DAE). The first step is to translate a hierarchical MODELICA model into a flat set of MODELICA statements, consisting on the set of equation and algorithm sections of all the used components. The resulting system of
Modelica

equations looks like the following:

\[ c := f_c(\text{rel}(v)) \]  
\[ m := f_m(v, c) \]  
\[ 0 := f_x(v, c) \]

where \( v := [\dot{x}; x; y; t; m; \text{pre}(m); p] \): \( p \) is the set of parameters and constant variables, while \( m \) is the set of discrete event variables. \( x \) and \( y \) are continuous variables and the difference between them is that \( x \) variables appear differentiated while \( y \) variables do not. \( \text{rel}(v) \) is the set of relations on variables in \( v \) and so \( c \) is the set of expressions in \textbf{if} statements including expressions coming from the conversion of \textbf{when} statements into \textbf{if}. A DAE solver will iterate in the following way:

- the equations \ref{4.4} are solved by assuming \( c \) and \( m \) constants, meaning that the system of equations is a continuous system of continuous variables;
- during integration of Equation \ref{4.4}, the conditions in \ref{4.2} are monitored. If a condition changes its status, an event is triggered at that specific time and the integration is halted.
- at an event instant equation \ref{4.3} is a mixed set of algebraic equations which is solved for the Real, Boolean and Integer unknowns;
- after the event is processed, the integration is restarted with \ref{4.2}.

4.4 Discussion

\textsc{modelica} is an object oriented language for mathematical programming. Object orientation is well understood in the software community and is certainly a well accepted programming paradigm.
4.4 Discussion

The language is very clean and there are important features that make it extremely easy to build models. First of all non-causal modeling allows designers to directly write model equations into the language syntax without any change. Designers don’t have to explicitly define dependent and independent variables. This saves the potential effort of solving equations or making different models depending on which quantities are computed and which are used to compute others.

Object orientation helps write reusable models. Inheritance gives the possibility of defining a basic set of equations that are common to many dynamical systems and then specialize a model depending on the real application.

Connections are simplified by a special keyword. In every physical system it is possible to distinguish quantities as through and across. MODELICA gives the possibility of declaring them with a special keyword. Connections are automatically translated into equations depending on the variables that are connected (either through or across).

MODELICA doesn’t specify the semantic of algebraic loops. The meaning is then let to a simulation tool that could for instance reject the program. For instance, this is how the DYMOLA simulator behaves.
Chapter 5

Sildex

Sildex is an integrated toolset for formally specifying and designing control and data-oriented real-time embedded systems. In particular, Sildex, which is produced and marketed by TNI-Valiosys, targets safety-critical embedded software applications whose failure may lead to disastrous consequences. It is used in the aerospace, automotive, energy, telecom, and defense industries.

5.1 The Sildex Approach

The Sildex approach is based on the synchronous paradigm and the synchronous language Signal (see next section). This provides a sound mathematical foundation which is key to guaranteeing the global coherence of the tool. Simulation, proof of properties, and code generation all rely on one single, rigorous, formal semantics, that is the semantics of Signal. The software aspects of the Signal language are transparent to the users of Sildex. In fact, they are not necessarily required to master the language to be able to manipulate Sildex design specifications. Instead, they can simply use the graphical design editors while benefiting from the formal semantics that lies in the background. The Sildex design environment puts emphasis on sound deterministic tool behavior and supports design hierarchy.
5.1 The Sildex Approach

In Sildex a unique specification diagram can be used to capture the internal coherence of the specification, simulate and formally verify it, and, finally, automatically derive executable embedded code for the system under design. In the specification of a design, each component appears as a graphical symbol showing its function. The body of primitive components is described directly in the SIGNAL language; hierarchical components, on the other hand, are represented as an inner diagram. Components are stored in libraries which can be created and enriched at will, thereby supporting design reuse. To describe each component, the users may choose from different styles: data-flow style, state machines, truth tables, and sequential function charts (SFC/Grafcets) [21, 41, 43]. They can import diagrams previously created under the SIMULINK tool through the SIMUSILD gateway available inside Sildex, or write components in the C language. Components are then wired together in a data-flow fashion.

Sildex offers two high-level features for validating a specification diagram. First, the integrated simulator allows the user to execute interactively the embedded code generated by the compiler and follow visually the evolution of the program’s state machines and data flows. To complement the functional validation provided by simulation, Sildex offers the capability of a formal proof mechanism of safety properties. Initially, such properties are often given as informal statements like “the engine should not start as long as seat belts are not fastened”. Then, after being expressed formally in the specification diagram, they can be checked via formal proof instead of exhaustive simulation. The proof mechanism is automatic and, therefore, not subject to errors due to improper use of an interactive proof device.
5.2 Synchronous Paradigm and Synchronous Languages

Synchronous programming languages like Esterel, Lustre, and Signal represent powerful tools for the specification of complex real-time embedded systems because they allow to combine the simplicity of the synchronous assumption with the power of concurrency in functional specification [26, 27, 74, 75]. They are synchronous systems with particular properties and for this reason, they are often considered a model of computation in addition to the generic synchronous model. The synchronous programming model can be expressed by the following “pseudo-mathematical” statements [24, 25]:

\[ P \equiv R^\omega \]
\[ P_1||P_2 \equiv (R_1 \land R_2)^\omega \]

where \( P, P_1, P_2 \) denote synchronous programs, \( R, R_1, R_2 \) denote the sets of all the possible reactions of the corresponding programs, and the superscript \( \omega \) indicates non-terminating iterations. The first expression interprets the essence of the synchronous assumption: a synchronous program \( P \) evolves according to an infinite sequence of successive atomic reactions. At each reaction, the program variables may or may not present a value. The second expression defines the parallel composition of two components as the conjunction of the reactions for each component. This implies that communication among components is performed via instantaneous broadcast.

An important feature offered by the synchronous programming model is the ability of taking decisions based on the absence of a value for a variable at a given reaction, i.e., in synchronous systems absence can be sensed. The notion of clock of a variable is introduced as a Boolean meta-variable tracking the absence/presence of a value for the corresponding variable. Variables that are always present simultaneously are said to have the same clock, so that clocks can be seen as equivalence classes of simultaneously-present variables. In the sequel, we focus our attention on Signal, which is the
declarative language representing the basis of SILDEX [26]. Besides parallel composition, SIGNAL’s main operators are the followings:

- statement $c := a \ op \ b$, where $op$ denotes a generic logic or arithmetic operator, defines not only that the values of $c$ are function of those of $a$ and $b$, but also that the three variables have the same clock;

- statement $c := a$\$k$, where $k$ is a positive integer constant, specifies both that $c$ and $a$ have the same clock and that at the $n$-th reaction when the two signals are present, the value of $c$ is equal to the value held by $a$ at the $(n-k)$-th reaction;

- statement $c := a$ default $b$ specifies that variable $c$ is present at every reaction where either $a$ or $b$ is present while taking the value of $b$ only if $a$ is not present (oversampling);

- statement $c := a$ when $b$ specifies that variable $c$ is present (taking the value of $a$) only when both $a$ is present and the Boolean condition expressed by variable $b$ is true (undersampling).

While the first two statements are single-clock, the last two are multi-clock. Additional operators are available to directly relate the variable clocks: for instance, statement $c \land= a$ constraints variables $c$ and $a$ to have the same clock, without relating the values that they assume. The SIGNAL compiler uses the clock calculus to statically analyze every program statement, identify the structure of the clock of each variable, and schedule the overall computation. The compiler rejects the program when it detects that the collection of its statements as a whole contains clock constraint violations. A public domain SIGNAL compiler is available as part of the POLYCHRONY toolset [72] at www.irisa.fr/espresso/Polychrony.
5.3 The Sildex Integrated Toolset

The Sildex integrated toolset covers most of the embedded software development life cycle including: hierarchical graphical specification, validation of specifications, conception of final system, embedded automatic distributed code generation, formal verification, and testing.

The toolset features an editor for each of the supported formalisms: data-flow, state machines, truth tables, SFCs. Besides editing new components and import Simulink/Stateflow diagrams, users can rely on a library of pre-designed components.

For validation purposes Sildex provides both an efficient interactive simulator and formal proof mechanisms of safety properties. For embedded software synthesis Sildex features fully-automated, customizable generators of executable C and Ada code.

Finally, the toolset provides support for automatic generation of the software documentation.

5.4 Discussion

Unlike the Simulink/Stateflow design environment, the Sildex integrated toolset relies on sound mathematical foundations, namely the one provided by the semantics of SIGNAL. Such foundation not only enables the introduction of tools for the automatic proof of formal properties, but it also represents a guarantee of consistency and robustness for the overall behavior of the system. Another important feature of Sildex is the automatic generation of embedded code from high-level design specification. This also contributes to the safety of the design approach as users can abstract from low-level design details while focusing on the validation at higher levels of abstraction. On the other hand, Sildex current focus is on embedded software design and the tool lacks the capability of modeling generic hybrid systems.
Chapter 6

HyVisual

The Hybrid System Visual Modeler (HyVisual) is a block-diagram editor and simulator for continuous-time dynamical systems and hybrid system [85]. HyVisual is built on top of Ptolemy II [55, 99], a framework that supports the construction of such domain specific tools, and can be freely downloaded from http://ptolemy.eecs.berkeley.edu.

6.1 HyVisual Graphical Syntax

Like in Ptolemy II, a HyVisual model is built starting from a set of library actors. An actor is a block with typed ports and parameters. Output ports can be connected to input ports by means of relations. The type of the output port must be greater than or equal to the type of the input port, where $t_1 \geq t_2$ if a variable of type $t_1$ can be converted into $t_2$ without loss of information. Users don’t necessarily have to specify all the relations and the actors library is rich enough to model most practical systems. Naturally, users have the option to build new actors and redefine relations.

A composite actor contains interconnection of other actors. It represents a subsystem encapsulated in an actor. This is a way of representing hierarchy.
Finally, a *modal model* is an actor that has modes of operation. A modal model is captured as a finite state machine that can be specified by drawing bubbles (states) and connecting them through arcs (transitions). Each bubble can be refined into a continuous time system representing a dynamical system or into another finite state machine.

A hybrid system can be described in **HyVisual** as follows. A modal model is instantiated and its ports are configured. The finite state machine that describes its mode of operations is represented as a graph. Each state has a name and each transition is characterized by the following elements:

- **guard expression**: it is a boolean expression involving inputs and outputs of the modal model as well as state variables;

- **output actions**: it is an assignment of values to the output ports;

- **set actions**: it is an assignment of values to the state variables;

- **reset**: it is a boolean value (either zero or one);

- **preemptive**: it is a boolean value (either zero or one);

Each state can be refined into a dynamical system or into another finite state machine. The user describes a dynamical system by using actors from the built-in libraries. These includes actors for standard computation (like addition, multiplication, etc.), as well as actors to model continuous dynamics (the *dynamics* library) like Integrator, LaplaceTransferFunction, LinearStateSpace, DifferentialSystem. When a modal model is created, its ports are propagated to the state machine diagram and to all its refinements.

A **HyVisual** model is saved in XML format. The XML file is a text file describing the actors used in the model, their port and parameter configuration, and their graphical properties (shape and position).
6.2 HyVISUAL Semantics

After a model is graphically built in HyVISUAL, it can be compiled into an executable form and simulated. The compiler controls the consistency of the model and, in particular, it performs type checking. The general rule follows the semantics of the JAVA language. HyVISUAL performs an automatic type conversion when there is no loss of information. For instance, an integer port can be connected to a double port and the data is automatically cast from integer to double. When a conversion is not possible, HyVISUAL reports an error message.

As in all hybrid modeling environment, HyVISUAL can mix discrete signals and continuous signals. A discrete signal consists of events on a timeline. An event has a tag and a value. If a discrete signal is examined at a time where it has no events, it will present no value. Hence, the semantics of discrete signals in HyVISUAL is different from MODELICA where a discrete signal maintains a constant value between two consecutive events.

Continuous actors cannot read discrete signals. The users must manage explicitly the communication between discrete and continuous domains using actors from the following libraries:

- **todiscrete**: it contains *event generators, level crossing detectors, samplers and threshold monitor*.

- **tocontinuos**: it contains *zero-order hold and first-order hold actors*.

A hybrid system model is described starting from a modal model. A modal model has modes of operations represented by the states of a state machines. Each state can be refined into a continuous time model (describing the equations of a dynamical system) or another finite state machine. While in the current state, its refinement is executed by a refinement solver which is a continuous time solver. In the current release (HyVISUAL 3.0.2), the default solver is a Runge-Kutta with a variable time step. In the case where a model has an actor checking whether a continuous variable is equal to a
certain value, then the time step is dynamically adjusted to determine the
crossing point within a certain error tolerance. Those special actors, such as
the `LevelCrossingDetector` actor, can be found in the `todiscrete` library.

Transitions between states are described as arcs. A transition has a
boolean-valued guard that has to evaluate to true in order for that transition
to be enabled. A transition can be refined into a netlist of components. The
semantics associated with a transition refinement is that it is executed when
if the guard has evaluated to true, for one iteration. The definition of and
iteration depends on the domain in which the netlist is described.

A transition has `output actions` and `set actions`. Output actions don’t
affect states while set actions do.

Finally there are two check boxes: `preemptive` and `reset`. When reset
is checked, the model that forms the refinement of the destination state is
executed from its initial conditions. If the box is not checked, then that
model is executed from whatever state it was last in.

**HYVISUAL** does not allow non-deterministic models. If at some point
there are more that one enabled transitions (with guards evaluating to true)
an error message pops up and the execution is stopped.

The execution semantics is defined as follows [85]:

- for any transitions out of the current state for which preemptive is true,
  the guard is evaluated. If exactly one such guard evaluates to true,
  then that transition is chosen. The output actions of the transition are
  executed, and the refinements of the transition (if any) are executed,
  followed by the set actions;

- if no preemptive transition evaluated to true, then the refinement of
  the current state, if there is one, is evaluated at the current time step;

- once the refinement has been evaluated (and it has possibly updated
  its output values), the guard expressions on all the outgoing transi-
  tions of the current state are evaluated. If none is true, the execution
6.3 Examples

is complete. If one is true, then that transition is taken. If more than
one is true, then an exception is thrown (the state machine must be
nondeterministic). “To take a transition” means that its output ac-
tions are executed, its refinements (if any) are executed, and its set
actions are executed;

• if reset is true on a transition that is taken, then the refinement of the
destination mode (if there is one) is initialized.

6.3 Examples

This section presents some examples of how HyVisual can be used. First,
we discuss the same circuit example that we also modeled with Modelica as
described in Section 4.3. Using Modelica we were able to define two pins
components and then extend them to model resistor and capacitor. Using
connectors and defining current as a flow variable, we were also able to
connect components as we see them in the block diagram. In HyVisual we
need to take a different approach.

First we create two distinct composite actors, one for a resistor compo-
nent and one for a capacitor component. Since HyVisual does not allow
non-causal modeling, we must decide what the inputs and outputs of a model
are. We build a resistor component whose input is a voltage and whose out-
put is a current. On the other hand, our capacitor will receive a current
as input and will compute the voltage across its terminals as output. The
two composite actors are shown in Figure 6.1. The resistor has a parameter
$R$ representing the resistance value. The input voltage is scaled by $1/R$
to
give the output current. The capacitor has a parameter $C$ representing its
capacitance. The equation implemented by the graphical representation is
$v(t) = 1/C \int i(t)dt$.

Figure 6.2 shows the finite state machine of the modal model and its
state refinement (except for the initial state marked as “Init”). The figure
Figure 6.1: Composite actors for a resistor and a capacitor.

Figure 6.2: Finite state machines and state refinement of the circuit example shows all the steps from the top level modal model to the state refinements.
6.4 Discussion

The equations described by the refinement are:

\[ VC(t) = \frac{1}{C} \int i_R(t) dt \] (6.1)

\[ i_R(t) = \frac{1}{R} v_R(t) \] (6.2)

\[ v_R(t) = \text{Source} - VC(t) \] (6.3)

where \( i_R \) and \( v_R \) are the current through and the voltage across the resistor. Equation 6.1 is implemented by the CurrentControlledCapacitor actor, Equation 6.2 is implemented by the VoltageControlledResistor actor and Equation (6.3) is implemented by the AddSubtract actor. Figure 6.3 shows the simulation result.

6.4 Discussion

HyVisual is a graphical environment for modeling hybrid systems. Graphical representations have the advantage of being intuitive and easy to use. There is a rich library of components making the language expressive enough to model hybrid systems. Type checking and inference are desirable features in designing large systems, because they help the users focus on the structure of the system rather than on the interface of components. Implementation of hierarchy in HyVisual is very clean and gives the user the possibility of encapsulating subsystems in larger blocks. Furthermore, state machines can be hierarchical in the sense that a state can be refined in other state machines. This feature of grouping states is very important when dealing with systems having a large state-space.

It is important to stress that state and transition refinements can be arbitrary PtolemyII models. This is a contrast with Simulink, for instance, where the states of Stateflow are atomic objects, and the control they exercise over
Finally, the entire design is stored in XML format, which can be easily transformed in other format by appropriate schemas (XSTL).

One disadvantage of HyVisual is the inability to express non-determinism. Non-determinism can be quite useful, particularly at the early stages of the design process where specifications are usually incomplete. While it is generally easy to express a system in a graphical format using HyVisual, it is difficult to derive the equations governing the hybrid system starting from the graphical representation.
Chapter 7

Scicos

SCICOS (Scilab Connected Object Simulator) is a Scilab package for modeling and simulation of dynamical systems including both continuous and discrete time subsystems [108]. Scilab (Scientific Laboratory) is a scientific software package for numerical computations that provides a powerful open-computing environment for engineering and scientific applications [67]. Since 1990 Scilab has been developed by researchers from INRIA and ENPC. In May 2003 the newly created Scilab Consortium took over maintenance and development of Scilab. Since 1994 Scilab has been distributed freely via the Internet and used in educational and industrial environments around the world. Scicos has been developed also at INRIA and is freely available for download at “http://www.scicos.org”. Roughly speaking, Scilab can be seen as a free MATLAB while Scicos is similar to Simulink.

Scicos users can build models of hybrid systems by composing functional blocks from a predefined library (as well as newly-defined blocks) and simulate them. This is done within a graphical editor. Additionally, users can generate executable C code implementing the functionality of some sub-system in the original hybrid system. This is limited to discrete time sub-systems, i.e. subsystems that don’t include continuous-time blocks. The main application here is embedded control: continuous blocks can be used
to model the physical environment while the discrete subsystems specify the functionality of the controller. After simulating and refining the design of the controller, the user can generate the C code and run it on the target hardware architecture. Finally, for the important case of distributed real-time applications, the users can rely on the SCICOS-SYNDEX interface [53] to generate and deploy executable code on multiprocessors architectures. SYNDEX is a system-level CAD software for distributed real-time embedded systems that has been designed and developed at INRIA and is freely available at “www-rocq.inria.fr/syndex”.

7.1 Signals and Blocks

A system is modeled in SCICOS by assembling functional components called blocks that interact by means of signals. A signal $x$ in SCICOS is a pair $\{x(t), T\}$, where $x(t)$ denotes that $x$ is function of time and $T$ is the associated activation time set on which signal $x$ can potentially evolve and
7.1 Signals and Blocks

change its value \[22\]. The activation time set is the union of time intervals and isolated points called events. In fact, a generic signal in Scicos can be the result of operating on both continuous (time intervals) and discrete (time events) signals. Outside its activation time set, a Scicos signal is constrained to remain constant as illustrated in Figure 7.1, which shows the evolution of a signal \(x\) having an hybrid nature (both continuous and discrete).

Activation time sets are used in Scicos in the same way as clocks are used in Signal (see Section 5.2), namely as a type checking mechanism. For instance, two signals can be constrained to have identical time sets and, in general, the various Scicos signal operators induce relations between the corresponding time sets. In general, given a generic binary operator \(f\), the activation time set of the resulting signal is the union of the activation time sets of its operands, i.e.:

\[
f(\{x_1(t), T_1\}, \{x_2(t), T_2\} ) = \{ f(x_1(t), x_2(t)), (T_1 \cup T_2) \}
\]

It is possible to reason formally on the time sets of Scicos signals as it is the case for the clocks of Signal variables \(^1\). Hence, Scicos users have a sound basis for tasks like design optimization and scheduling analysis.

Each Scicos operation is associated to a Scicos block and the activation times of a signal correspond to the activation times of the block that generates it. Figure 7.2 illustrates a generic Scicos block. This can present ports associated to four different signal types: regular input, regular output, activation (event) input, activation (event) output. By convention these ports are placed respectively on the left, right, top, and bottom side of the block. In fact, the set of signals in Scicos is partitioned in two subsets: regular signals and activation signals. Activation signals are also called event signals or impulses. Regular signals are used to exchange data among blocks, while activation signals carry control information. Regular inputs are linked to regular outputs via regular paths, while activation inputs are

\(^1\)Notice however that Scicos’s model of computation is not the same as Signal’s one.
Figure 7.2: A generic Scicos block and its I/O signals.

linked to activation outputs via activation paths. Regular paths carry piece-wise right-continuous functions of time whereas event paths transmit timing information concerning discrete events (impulses). In particular, an event signal specifies the time when the blocks connected to the output event port generating the event signal are updated according to the internal relations of the block (see Section 7.2).

An activation signal causes the block to evaluate its outputs and new internal states as a function of its inputs and previous internal states. A block with no input activation port is permanently active (time-dependent block). The output signals inherit their activation times set from the union of the activation times of the input signals of the generating block. In turn, they can be used to drive other blocks. The signals leaving the output activation ports are activation signals generated by the block. For instance, a CLOCK BLOCK may generate a periodic activation signal that can be connected to the input of a SCOPE BLOCK to control the sampling of the inputs signals of the latter [108].
There are two general types of blocks: \textit{basic blocks} and \textit{super blocks}. Super blocks are obtained as the hierarchical composition of basic blocks and other super blocks. Scicos comes with a library of more than 70 basic blocks\cite{108}. Additionally, the users can build new basic blocks by defining an \textit{interfacing function} and a \textit{computational function} for each of them. The former is always a Scilab function, while the latter can also be written in C or Fortran (with important performance gain from a simulation viewpoint). Besides defining the graphical aspect of the block, the interfacing function allows users to define the number and types of ports and to initialize the state and parameters of the block. The computational function specifies the dynamic behavior of the block through a set of tasks and is called by the Scicos simulator that controls their execution. Depending on the type of the block and the directive of the simulator, the invocation of a computational function may result in various actions like evaluation of new outputs, state update, or computation of the state derivative.

\section*{7.2 Basic Blocks}

There are four types of basic blocks: \textit{continuous}, \textit{discrete}, \textit{zero-crossing}, and \textit{synchro}.

\subsection*{7.2.1 Continuous Basic Blocks (CBB)}

A CBB can have both regular input (output) ports and event input (output) ports. CBBs can model more than just continuous dynamics systems. A CBB can have a continuous state $x$ and a discrete state $z$. Let the vector function $u$ denote the regular inputs and $y$ the regular outputs. Then a CBB imposes the following relations:

\begin{align*}
    \dot{x} &= f(t, x, z, u, p) \\
    y &= h(t, x, z, u, p)
\end{align*}
where \( f \) and \( h \) are block specific functions, and \( p \) is a vector of constant parameters. The above relation represents two constraints that are imposed by the CBB as long as no events (impulses) arrive on its event input ports. An event input can cause a jump in the states of the CBB. Let’s say one or more events arrive on CBBs event ports at time \( t_e \), then the states jump according to the following equations:

\[
\begin{align*}
x &= g_c(t_e, x(t_e^-), z(t_e^-), u(t_e^-), p, n_{eprt}) \\
y &= g_d(t_e, x(t_e^-), z(t_e^-), u(t_e^-), p, n_{eprt})
\end{align*}
\]

where \( g_c \) and \( g_d \) are block specific functions, \( n_{eprt} \) designates the ports through which the events have arrived, and \( z(t_e^-) \) is the previous value of the discrete state \( z \) (which remains constant between any two successive events). Finally, CBBs can generate event signals on their event output ports. These events can only be scheduled at the arrival of an input event. If an event has arrived at time \( t_e \), the time of each output event is generated according to

\[
t_{evo} = k(t_e, z(t_e), u(t_e), p, n_{eprt})
\]

for a block specific function \( k \) and where \( t_{evo} \) is a vector of time, each entry of which corresponds to one event output port. Normally all the elements of \( t_{evo} \) are larger than \( t_e \). If an element is less than \( t_e \), it simply means the absence of an output event signal on the corresponding event output port. Notice that setting \( "t_{evo} = t" \) should be avoided because the resulting causality structure is ambiguous. Also, notice that setting \( "t_{evo} = t" \) does not mean that the output event is synchronized with the input event because two events can have the same time without being synchronized. Event generation can also be pre-scheduled by setting the corresponding initial firing in the CBB. Only one output event can be scheduled on each output event port (both at the beginning and in the course of the simulation). In other words, by the time a new event is ready to be scheduled, the old one must have been fired already. This is natural because the register that contains
7.2 Basic Blocks

the firing schedule of a block should be considered as part of the state, having
dimension equal to the number of output event ports. Another interpreta-
tion is that as long as the previously scheduled event has not been fired yet,
the corresponding output port is considered busy, meaning it cannot accept
a new event scheduling. If the simulator encounters such a conflict, it stops
and returns the event conflict error message [108].

7.2.2 Discrete Basic Blocks (DBB)

A CBB permanently monitors its input ports and continuously updates its
output ports and continuous state. In contrast, DBBs only act when they
receive an input event and their actions are instantaneous. DBBs can have
both regular and event input and output ports, but they must have at least
one event input port. DBBs can model discrete dynamics systems. A DBB
can have a discrete state $z$ but no continuous state. Upon the arrival of
events at time $t_e$, the state and the outputs of a DBB change as follows

\[
  z = f_d(t_e, z(t_e^-), u(t_e^-), p, n_{evprt})
\]
\[
y = g_d(t_e, z, u(t_e), p)
\]

where $f_d$ and $h_d$ are block specific functions. The regular output $y$ remains
constant between any two successive events. In fact, the output $y$ and the
state $z$ are piece-wise constant, right-continuous functions of time. Like
CBBs, DBBs can generate output events according to a specific function
$k$ and their events can be pre-scheduled via initial firing. The difference
between a CBB and a DBB is that a DBB cannot have a continuous state
and that its outputs remain constant between two events. Although in
theory CBBs subsume DBBs, specifying a block as a DBB corresponds to
give a directive to the simulator that can optimize its performance because
it knows that the outputs of this block remain constant between events.
Note that the regular output signal of a DBB is always piece-wise constant.
Being piece-wise constant does not necessarily imply that a signal is discrete,
for example the output of an integrator (which is a CBB with continuous state) can, in some special cases, be constant. However, signals that are piece-wise constant can be identified based solely on the basic properties of the blocks that generate them. In particular, in Scicos, every regular output signal of a DBB is discrete and every regular output signal of a stateless time invariant CBB receiving only discrete signals on its inputs is also discrete. Thus, the discrete nature of signals in a model can be specified statically. Again, the Scicos compiler relies on this information to optimize the performance of the Scicos simulator.

7.2.3 Zero Crossing Basic Blocks (ZCBB)

ZCBBs have regular inputs and event outputs but no regular outputs, or event inputs. A ZCBB can generate event outputs only if at least one of the regular inputs crosses zero (i.e., it changes sign). In such a case, the generation of the event, and its timing, can depend on the combination of the inputs which have crossed zero and the signs of the inputs (just before the crossing occurs). The simplest example of a Surface Crossing Basic Block is the zcross [108]. This block generates an event if all the inputs cross simultaneously zero. Inputs of ZCBBs can start off at zero, but cannot remain equal to zero during the simulation. This is considered an ambiguous state and is declared as an error. Similarly the input of a ZCBB should not jump across zero. If it does, the crossing may or may not be detected. ZCBBs cannot be modeled as CBBs or DBBs because in these blocks, no output event can be generated unless an input event has arrived beforehand.

7.2.4 Synchro Basic Blocks (SBB)

SBBs are the only blocks that generate output events that are synchronized with their input events. These blocks have a unique event input port, a unique (possibly vector) regular input, no state, no parameters, and two or more event output ports. Depending on the value of the regular input, the
7.2 Basic Blocks

Figure 7.3: Modeling embedded control as a hybrid system.

incoming event input is routed to one of the event output ports. SBBs are used for routing and under-sampling event signals. Typical examples are the EVENT SELECT BLOCK and the IF-THEN-ELSE BLOCK [108].

7.2.5 Synchronization

In Scicos if two event signals have the same time, they are not necessarily synchronized. In other words, one is fired just before or just after the other but not “at the same time”. Two event signals can be synchronized only when they can be traced back to a common origin (a single output event port) through event paths, event additions, event splits, and SBBs alone. In particular, a basic block cannot have two synchronized output event ports. This is possible however for Super Blocks like the 2-FREQ CLOCK BLOCK [108].
7.3 Interface from Scicos to SynDEx

SynDEx is a system level CAD software for rapid prototyping and optimizing the implementation of distributed real-time embedded applications onto “multi-component” architectures. It is based on the “algorithm-architecture adequation” (AAA) methodology [68, 125]. The AAA methodology aims to find the best matching between an algorithm and architecture while satisfying real-time constraints. This is formalized in terms of graphs transformations. The algorithm is specified with a data-flow graph while the architecture is capture via a multiprocessor hyper-graph. Then, an implementation is derived by distributing and scheduling the former on the latter. The result of the graphs transformations is an optimized Synchronized Distributed Executive (a SynDEx), which is automatically built from a library of architecture dependent executive primitives composing the executive kernel [68]. These primitives support boot-loading, memory allocation, interprocessor communications, sequencing of user supplied computation functions and of interprocessor communications, and inter-sequences synchronizations. The users are provided with a library of executive kernels for various supported processors, while kernel for other processors can be ported from the existing ones.

Based on this methodology, SynDEx enables rapid prototyping of complex distributed real-time embedded applications. This is centered on automatic code generation, which is performed in three steps:

1. implementation onto a single-processor workstation for simulation;

2. implementation onto a multi-processor system in order to study parallelism benefits and accelerate simulation;

3. real-time execution on the targeted multi-component architecture which may include programmable components (processors) as well as non-programmable components (ASICs).
### 7.3 Interface from Scicos to SynDEx

Figure 7.4: Scicos block diagram of the RC circuit example

The main feature of the SynDEx software is the seamless environment guiding the user from the specification level (functional specification, distributed hardware specifications, real-time and embedding constraints) to the distributed real-time embedded code level, through (multi-)processor simulations. In particular, it allows to automatically generate, distribute and schedule real-time embedded code.

The Scicos-SynDex interface [53] allows users to pair up Scicos and SynDex, thereby deriving a design flow distributed real-time embedded control applications that leverages the hybrid systems approach. The users can model an embedded control application in Scicos as it is sketched in Figure 7.3: a model for the physical plant (the environment) is obtained using continuous-time blocks while the controller is designed by assembling discrete-time blocks. The users can perform the “high-level” simulation of the whole hybrid system to reach a first-cut design of the controller. Then, the discrete subsystem modeling the controller is transfered into Syn-
DEX via the provided interface to generate the embedded code for the targeted distributed architecture. This step is simplified by the following facts: (1) SCICOS and SYNDEX share the same model of computation for the discrete subsystem (a data flow graph) and (2) the I/O interface of the functional discrete blocks is the same\(^2\). Also, SYNDEX tries to take advantage of the parallelism intrinsically captured by the data flow model to match the parallelism offered by the target architecture, thereby obtaining an implementation that satisfy the real-time constraints. Notice that the interface has been specifically developed for this kind of applications and does not support the translation of continuous-time basic blocks and zero-crossing basic blocks.

\(^2\)In fact, it may be the case sometimes that a single SCICOS block is translated into a group of SYNDEX blocks. For further details see [53].
7.4 Example

In this section we show how to model the simple RC circuit in Scicos. The circuit diagram is shown in Figure 7.4. Scicos does not provide a way of specifying state machines to drive the changing of modes in a hybrid system. Everything must be specified in a hierarchical diagram by using the basic building blocks. In this example, we need two threshold detectors to detect when the capacitor voltage crosses the values of $4V$ and $1V$. The corresponding Scicos block diagram is shown in 7.4. The output of the integrator (represented by the block $1/s$) is the capacitor voltage. Its value is compared with the two values 1 and 4. When the voltage swings from a value less than 4 to a value greater than 4, then the -to+ block will generate an event. This event switches the selector output from 5 to $-5$. Then, the capacitor voltage decreases until it reaches a value equal to 1. At this point the +to- block generates an event switching the selector output back to 5. Figure 7.5 reports the simulation results and shows that the simulation step is not properly adjusted to detect the event exact time.

7.5 Discussion

The block diagram of Figure 7.4 is very similar to the HyVisual block diagram shown in Figure 6.2. Both HyVisual and Scicos are based on the composition of causal blocks. While communication between continuous and discrete blocks in HyVisual is explicitly declared by the user through the instantiation of special communication blocks, a discrete event in Scicos hybrid blocks can update the continuous state.

An important difference is that Scicos does not have a way of specifying a state machine representing the hybrid automata but the state machine has to be explicitly implemented using a composition of blocks. HyVisual instead defines states and transitions. In our opinion this is an important feature that cannot be missing in a hybrid systems modeling environment.
Part III

Tools for Formal Verification
Chapter 8

Charon

CHARON, an acronym for coordinated control, hierarchical design, analysis and run-time monitoring of hybrid systems, is a high-level language for modular specification of multiple, interacting hybrid systems that has been developed at the University of Pennsylvania [3, 4, 5, 8]. CHARON is based on the notions of agent and mode and supports hierarchical modeling both at the architectural and behavioral level. For hierarchical description of the system architecture, CHARON provides the operations of instantiation, hiding, and parallel composition on agents, which can be used to build a complex agent from other agents. Modes are used to describe the discrete and continuous behaviors of an agent. For hierarchical description of the behavior of an agent, CHARON supports the operations of instantiation and nesting of modes. Furthermore, features such as weak preemption, history retention, and externally defined Java functions, facilitate the description of complex discrete behavior. Continuous behavior can be specified using differential as well as algebraic constraints, and invariants restricting the flow spaces, all of which can be declared at various levels of the hierarchy. The modular structure of the language is not merely syntactic, but is exploited by analysis tools, and is supported by a formal semantics with an accompanying compositional theory of modular refinement [6, 7].
8.1 The Charon Language

The Charon language enables specification of architectural as well as behavioral hierarchy and discrete as well as continuous activities.

8.1.1 Architectural Hierarchy

The architectural hierarchy reflects the composition of distinct processes working in parallel. In this framework the basic building block is represented by an agent. Agents model distinct components of the system whose executions are all active at the same time. They can be of two types:

- primitive
- composite

Primitive agents are the primitive types or basic building blocks of the architectural hierarchy. Composite agents are derived by parallel composition of primitive agents.

Other main operations supported by agents are variable hiding and variable renaming. The hiding operator makes a specified set of variables private or local, that is other agents cannot access private variables for read/write operations. Variable hiding implements encapsulation for data abstraction. Variable renaming is for supporting instantiation of distinct components having the same structure.

Agents communicate among themselves and with the external environment by means of shared variables, which represent input/output/state signals of the overall hybrid system.

8.1.2 Behavioral Hierarchy

The behavioral hierarchy is based on the sequential composition of system components acting sequentially in time. Such components are called modes. Modes represent the discrete and continuous behaviors of an agent. Each
8.1 The Charon Language

agent consists of one or more distinct modes which describe the flow of control inside an agent. Modes contain:

- Control points (entry points, exit points)
- Variables (private, input, output)
- Continuous dynamics
- Invariants
- Guards
- Nested submodes

Control points are where the flow of control enters or exits the given mode. The execution of the mode starts as soon as the flow of control enters in an entry location, and ends when it reaches an exit location. To each location (entry point or exit point) can be associated a guard condition, that is a rule or a set of rules enabling the control flow to actually go through a given entry or exit point, i.e. enabling the hybrid systems to make a jump or discrete transition.

As for agents, variables in a mode represent discrete or continuous signals. Input and output variables represent respectively input and output signals of the agent, while private variables either represent state signals not visible externally, or true auxiliary variables, such as those necessary to perform some functional computation.

Modes can be atomic or composite; composite modes contain nested submodes which can themselves be composite or atomic. Modes can have three types of constraints:

**invariants** : the flow of control can reside in a mode as long as an inequality condition, called the invariant, is satisfied (e.g. if $x$ and $y$ are two variables, an invariant can be of the form $|x - y| \leq \epsilon$). When invariants
are violated the flow of control must exit the active mode from one of its exit points.

**differential constraints**: this kind of constraints is for modeling continuous dynamics evolving in the current mode (e.g. by differential equations, like: \( \dot{x} = f(x, u) \)).

**algebraic constraints**: algebraic equations model resets of variables occurring during discrete transitions of the hybrid system. Variables’ values are reassigned using an algebraic expression, such as: \( y = g(x, u) \).

### 8.2 The Syntax

Agents and modes are both represented as tuples. If \( T = (t_1, \ldots, t_n) \) is a tuple, we denote the element \( t_i \) of \( T \) by \( T.t_i \). This notation can be extended to collection of tuples, so that if \( ST \) is a set of tuples, then:

\[
ST.t_i = \bigcup_{T \in ST} \{T.t_i\}.
\]

Variables should be formally distinct from their valuations: given a set \( V \) of variables a *valuation* is a function mapping variables in \( V \) to their respective values. We denote by \( Q_V \) the set of all possible valuations over \( V \). If \( s \) is a valuation of variables in \( V \) and \( W \subseteq V \), then \( s[W] \) is the restriction of the valuation \( s \) to variables in \( W \).

Continuous-time behaviors of modes is modeled by flows. A flow is a differentiable function \( f : [0, \delta] \to Q_V \), where \( \delta \) is called the *duration* of the flow.

**Modes.** A mode is a tuple \((E, X, V, S, M, Cons, T)\) where:

- \( E \) is a set of entry points and \( X \) is a set of exit points. There are two particular control points: a default entry \( de \in E \) and a default exit \( dx \in X \).
8.2 The Syntax

- $V$ is a set of variables, which can be analog or discrete (characterizing signals for flows and jumps of the hybrid system, respectively). Variables can also be local, their scope being limited only to the active mode, or global, if they can be accessed externally.

- $SM$ is a finite set of submodes.

- $Cons$ is a set of constraints, which can be of three types: differential, algebraic and invariant, as described in the previous section.

- $T$ is a set of transitions of the kind $(e, \alpha, x)$, where: $e \in E \cup SM.X$ and $x \in X \cup SM.E$; $\alpha$ is called the action associated to the current transition, and it updates variables (analog or discrete and global or local) when the mode undergoes the transition $T$.

From this definition it follows that atomic modes are those in which $SM = \emptyset$. Top-level modes are composite modes that are not contained in any other mode (they can only be contained in agents); they have only one non-default entry point and have no default exit points.

Agents. The syntax of agents is simpler than that of modes. An agent is formally defined as a tuple $(TM, V, I)$, where $V$ is a set of variables, $I$ is a set of initial states and $TM$ is a set of top level modes. The set of variables $V$ results from the disjoint union of the set of global variables $V_g$ and local variables $V_l$; formally: $V = V_g \cup V_l$ with $V_g \cap V_l = \emptyset$. The elements of an agent can be accessed through the “dot” operator $;$; for example $A.V_g$ is the set of global variables of the agent $A$.

Intuitively, top-level modes in $TM$ describe the behavior (i.e. execution in time) of the agent. As for modes, variables in agents can be local or global. The set $I$ of initial states can specify particular initialization of variables in the agent. Primitive agents are those having only one top-level mode, while composite agents contain several top-level modes and can be obtained by the parallel composition of primitive types.
8.3 The Semantics

Modes can exhibit both a continuous or discrete behavior, but not at the same time: this implies that a mode undergoes a sequence of jumps (discrete transitions) and flows (continuous executions). During a flow the mode follows a continuous trajectory subject to the corresponding differential constraints. Moreover, as soon as the trajectory does not satisfy the invariant constraints anymore, the mode is forced to make a discrete transition.

A jump is a finite sequence of discrete transitions of submodes and transitions of the mode itself which are enabled by the corresponding guards. Any discrete transition starts in the current active state of the mode and terminates as soon as either a regular exit point is reached or the mode yields control to its external environment via one of its default exit locations.

The execution of an agent can be derived from those of its top-level modes. A primitive agent has a single top-level mode, while composite agents have several top-level modes (each possibly containing submodes) and results from the parallel composition of other agents. Execution trajectories start from the specified set of initial states, and consist of a sequence of flows interleaved with jumps, defined by the modes associated to the agent.

The operators defined on agents are the following:

- **Variable hiding.** The hiding operator makes a set of variables in an agent private. This means that their scope remains within the agent and they cannot be accessible from the external environment.

- **Variable renaming.** The renaming operator makes a replacement of a set of variables inside an agent with another set of variables. This is useful for interfacing the agent with its external environment (i.e. with other agents).

- **Parallel composition.** The parallel composition $A_1 || A_2$ of the two agents $A_1$ and $A_2$ is another agent $A$ defined by the following relations:
8.4 Compositionality Results

- \( A.TM = A_1.TM \cup A_2.TM \)
- \( A.V_g = A_1.V_g \cup A_2.V_g \) and \( A.V_l = A_1.V_l \cup A_2.V_l \)
- if \( s \in A.I \) then \( s[A_1.V] \in A_1.I \) and \( s[A_2.V] \in A_2.I \)

8.4 Compositionality Results

The semantics of Charon is compositional, meaning that the semantics of one of its components (possibly the whole hybrid system) is entirely specified in terms of the semantics of its subcomponents. Such a compositionality result holds for both agents and modes. Indeed, the set of traces of a given mode is determined by the definition of the mode itself and by the semantics of its submodes. For a composite agent the set of trace can be reconstructed from traces of its top-level modes. We make this statement precise in the discussion below.

Compositionality in Charon is achieved by two distinct features:

1. mode refinement;
2. agents compositionality.

Mode refinement. A mode \( M \) is said to refine another mode \( N \) if it has the same global variables and control points and, moreover, every trace of \( M \) is a trace of \( N \).

Agents compositionality. Compositionality results holding for modes can be naturally extended to agents, given that an agent is basically a collection of modes with synchronized flows and interleaving jumps. In particular agent operators are compositional with respect to refinement.

The refinement relation can only be applied when two distinct agents are compatible. An agent \( A \) and an agent \( B \) are said to be compatible if \( A.V_g = B.V_g \). Hence we say that an agent \( A \) refines a compatible agent \( B \), and we denote it by \( A \preceq B \), if \( L_A \subseteq L_B \).
8.5 The CHARON Toolkit

CHARON is currently implemented and distributed in a toolkit, which includes several tools for the specification, development, analysis and simulation of hybrid systems. The CHARON toolkit is entirely written in JAVA and features:

- a graphical user interface;
- a visual input language (similar to STATEFLOW);
- an embedded type-checker;
- a complete simulator.

The graphical input editor converts the specified model into CHARON source code, also using an intermediate XML format.

The plotter is based on a package from the modeling tool Ptolemy, developed at U.C. Berkeley. It allows the visualization of system’s traces as generated by the simulator.

The CHARON toolkit is also fully compatible with external programs written in JAVA; the simulator itself is an executable Java program.

The CHARON toolkit Version 1.0 is freely distributed and can be downloaded from http://www.cis.upenn.edu/mobies/charon

8.6 Examples

In this section we show how the CHARON toolkit can be used for describing and simulating hybrid systems. We use the same two examples as in Section 4.3: the bouncing ball example and the circuit example. Two methods can be followed to build a new model with CHARON. The first method is text-based while the second method is based on a graphical interface which allows the users to specify the entire system. We follow the first method to better exercise all the features that the language provides.
A model is written as a text file following the Charon syntax. The bouncing ball system is represented as an agent which has only one mode that we call BallTop. Even if a bouncing ball can be described by one mode only, we use two modes to describe the system. One mode, that we call up and another mode that we call down. The latter corresponds to the case where the ball is freely falling down while the former represent the case where the ball is going up after hitting the ground. The mode down is described as follows:

```charon
mode down() {
    exit hit;
    entry falling;
    readWrite analog real x;
    readWrite analog real a;
    readWrite analog real v;
    diff velocity { d(x) == v }
    diff acceleration { d(v) == a }
    inv fallinv { x > 0 }
}
```

The equation governing the dynamics are differential equation relating position, velocity, and acceleration. Each differential equation is declared by using the special keyword `diff`, then the equation name and, finally, the body of the equation within curly brackets.

Mode down is characterized by an invariant imposing the vertical position to be strictly positive. For the up mode the set of equations is the same but here the invariant constraints the velocity to be strictly positive. Mode BallTop has the instance of both up and down modes and also defines the transitions between these two:

```charon
mode BallTop() {
    private analog real x;
```
private analog real v;
private analog real a;
mode goingdown = down();
mode goingup = up();
trans maintrans from default
to goingdown
  when true do {a = -9.81;
    x = 1.0 }
trans fromdowntoup from goingdown.hit
to goingup.hit
  when (x <= 0) do {v = -0.8*v}
trans fromuptodown from goingup.falling
to goingdown.falling
  when (v <= 0) do {}
}

The first transition is a dummy transition going from the default state to the
instance of the down mode. Acceleration and position are initialized here.
This is not a canonical way of initializing function. In fact, CHARON provides
a special keyword init for this purpose. Also the acceleration is always
constant and could be modeled as a parameter. The BallTop mode has two
other transitions. The first one is enabled when the ball hits the ground
(x ≤ 0). This transition goes from the exit point of the instance of down
to the entry point of the instance of up. If the transition is taken then the
velocity is reset to its opposite, but the value is discounted to model energy
loss. The second transition goes back to the down state where the velocity
v ≤ 0, meaning that the ball is at its maximum reachable position for that
bounce. Finally, an agent just instantiates the BallTop mode:

agent BouncingBall() {
  mode top = BallTop();
}
8.6 Examples

Simulation results are shown in Figure 8.1. Even if the bouncing ball system is a classical example of Zeno automata, the simulation can proceed without any problem. Zeno behaviors should be detected and reported to the users, who might want to modify its code and refine the models to avoid them.

The second example that we show is the RC circuit. A continuous voltage generator is connected in series with the resistor and the capacitor. The output value of the voltage generator depends on the voltage across the capacitor. The first step is to define a Resistor component and a Capacitor component. A resistor has one mode of operation:

```agent Resistor (real R) {
    readWrite analog real ir;
    readWrite analog real vr;
    mode ResistorEqModeInst = ResistorEqMode(R)
}
mode ResistorEqMode(real R) {
```
ResistorEqMode has a parameter which is the value of the resistance. The only algebraic constraint that is fixed by the main mode is the relation between current and voltage in the resistor.

A capacitor is characterized in a similar way:

agent Capacitor (real C) {
    readWrite analog real ic;
    readWrite analog real vc;
    mode CapacitorEqModeInst = CapacitorEqMode(C)
}

mode CapacitorEqMode(real C) {
    readWrite analog real ic;
    readWrite analog real vc;
    diff ceq { d(vc) == ic/C}
}

The main difference is that the relation between current and voltage is now a differential equation. Finally, we model the circuit by instantiating both the capacitor and the resistor and by adding a mode that alternates between two submodes: one for charging the capacitor and one for discharging it.

agent Circuit () {
    private analog real dc;
    private analog real ir;
    private analog real vr;
    private analog real ic;
    private analog real vc;
    init { dc = 5.0; vc = 0.0}
8.6 Examples

agent ResistorInst = Resistor(10.0)
agent CapacitorInst = Capacitor(0.01)
agent CircuitModeContainerInst = CircuitModeContainer()
}

The circuit contains three agents: a resistor, a capacitor, and a third agent that is used only to connect the other two and to describe the mode switching. It seems indeed that the syntax of CHARON does not allow a mixed description of an agent using both sub-agents and modes.

CircuitModeContainer instantiates one mode which is called CircuitMode which in turn has two sub-modes Charging and Discharging.

agent CircuitModeContainer() {
    readWrite analog real dc;
    readWrite analog real vr;
    readWrite analog real ic;
    readWrite analog real ir;
    readWrite analog real vc;
    mode CircuitModeInst = CircuitMode()
}

mode CircuitMode () {
    readWrite analog real dc;
    readWrite analog real vr;
    readWrite analog real ic;
    readWrite analog real ir;
    readWrite analog real vc;
    alge { ic == ir}
    alge { vr == dc - vc }
    mode ChargingInst = Charging()
    mode DischargingInst = Discharging()
    trans deft from default to ChargingInst
when true do {}
trans disc from ChargingInst to DischargingInst
when (vc >= 4) do {}
trans ch from DischargingInst to ChargingInst
when (vc <= 1) do { }
}

mode Charging () {
  readWrite analog real dc;
  readWrite analog real vc;
  alge { dc == 5.0}
  inv chinv { vc < 4}
}

mode Discharging () {
  readWrite analog real dc;
  readWrite analog real vc;
  alge { dc == -5.0}
  inv chinv { vc > 1}
}

CircuitMode also establishes the connection between the capacitor and the resistor. A series connection is represented by an algebraic constraint forcing the current through the two components to be the same. The other algebraic constraint says that the voltage across the resistor is equal to the difference between the source voltage and the capacitor voltage. Finally there are two modes. During the Charging phase the source voltage is set to the value $5V$ until the capacitor voltage reaches the value of $4V$. This is what the invariant says (of course when the invariant is violated there is an output transition enabled). Similarly the discharging phase is characterized by a source value of $-5V$. 
8.6 Examples

Figure 8.2: Circuit example result

CircuitMode also describes the switching conditions between charging and discharging with a syntax similar to the bouncing ball example. Note that there are other modeling options. It is possible for instance to remove the algebraic equations in Charging and Discharging and use a set statement in the do section of the mode transition.

CHARON uses algebraic and differential constraints, respectively \( y = g(x, u) \) and \( \dot{x} = f(x, u) \). In these constraints the left-hand side is the unknown, while variables in the right-hand expression must be known. In fact, CHARON does not allow non causal modeling. Also there is no support for
causality detection. If we change for instance the algebraic equation \( ic = ir \) to \( ir = ic \) the simulation result is totally different. The use of readWrite variables should be avoided then since could lead to false conclusions.

Simulation results are shown in Figure 8.2, where the enlarged parts show what happens in a neighborhood of the switching events. Event-time detection is not precise but still good. Invariants seem to make the switching time estimation much better. This is mainly due to the fact that if a transition is enabled then it \textit{could} be taken, while if an invariant is not satisfied then a transition \textit{must} be taken. Invariants violation is detected at simulation time.
Chapter 9

CheckMate

CheckMate is a hybrid system verification toolbox for MATLAB that has been developed at Carnegie Mellon University. This chapter reviews how modeling and verification of hybrid systems is performed in this environment and it is based in particular on [122].

9.1 Main Features

CheckMate supports simulation and verification of a particular class of hybrid dynamic systems called threshold event-driven hybrid systems (TEDHS). A verification procedure for such system was proposed in [40]. In a TEDHS, the changes in the discrete state can occur only when continuous state variables encounter specified thresholds. Thresholds in the TEDHS model are hyperplanes. In the lingo of the general hybrid system model presented in Chapter 2, guards and invariants depend only on states linearly and are complementary, i.e., when invariants are not satisfied, an appropriate guard must be satisfied. This guarantees that when the system has to jump because the invariant is not satisfied at a state, there is a transition that it can take and hence, the behavior is non blocking.

Hybrid system models in CheckMate have continuous dynamics de-
scribed by standard differential state equations (possibly nonlinear), planar switching surfaces, and discrete dynamics modeled by finite state machines. The key theoretical concepts used in CheckMate are described in [39].

A very interesting feature of CheckMate is the use of standard industrial tools to enter the description of the hybrid systems to be analyzed and designed. CheckMate models are constructed using custom and standard Simulink and Stateflow blocks. The continuous state equations, parameters and specifications (the properties to be verified) are entered using the Simulink GUI and user-defined Matlab m-files. Specifications express properties of trajectories of the CheckMate model. The CheckMate verification function determines if the given specifications are true for all trajectories starting from a polyhedral set of initial continuous states and continuous ranges of parameter values. Note that the semantics of the design must be the one understood by CheckMate and for this reason, the tool uses the syntax of the Simulink environment but restricts its semantics so that a formal approach can be used.

Formal verification is a very expensive proposition for hybrid systems. In CheckMate, formal verification is performed on sets of states that can be reached from the initial conditions. Deriving the set of reachable states is computationally very hard even for linear time-invariant continuous-time systems. Hence there is a strong incentive for approximating the problem in a way that it is computationally feasible. In CheckMate, formal verification is performed using finite-state approximations known in the literature as quotient transition systems. The approximations are conservative in the sense that they capture all possible behaviors of the hybrid system. If the verification of an approximating automaton returns a positive result, the user is informed and the program terminates. If a negative result is found, the verification of the original hybrid system is inconclusive and the user is given the option to refine the current approximation and attempt the verification again. Properties of individual trajectories of the system can be
verified using the MATLAB simulation engine and the CHECKMATE validation tool.

9.2 The CHECKMATE Model

The hybrid system model understood by CHECKMATE is the threshold-event-driven hybrid system (TEDHSs), a class of hybrid automata [77]. A threshold-event-driven hybrid system consists of three types of subsystems:

- the switched continuous system (SCS),
- the threshold event generator (TEG), and
- the finite state machine (FSM).

The SCS is the continuous dynamics system that takes in the discrete-valued input $\sigma$ and produces its continuous state vector $x$ as the output. The continuous dynamics for $x$ evolve according to the differential equation or differential inclusions selected by the discrete input $\sigma$.

The output of the SCS is fed into the TEG, which produces an event when a component of the vector $x$ crosses a corresponding threshold from the specified direction (rising, falling, or both).

The event signals from the TEG drive the discrete transitions in the FSM whose output, in turn, drives the continuous dynamics of the SCS.

CHECKMATE converts the TEDHS into a polyhedral invariant hybrid automaton (PIHA). PIHA is a subclass of hybrid automata as presented in [77].

Recalling the definitions in Chapter 2, each discrete state in the hybrid automaton is called a location. Associated with each location is an invariant, the condition which the continuous state must satisfy while the hybrid automaton resides in that location, and the flow equation representing the continuous dynamics in that location. Transitions between locations are called edges. Each edge is labeled with guard and reset conditions on the
continuous state. The edge is enabled when the guard condition is satisfied. Upon the location transition, the values of the continuous state before and after the transition must satisfy the reset condition. In general, the analysis of hybrid automata can be very difficult. In CheckMate, attention is restricted to PIHA. Semi-formally, a PIHA is a hybrid automaton with the following restrictions:

- the continuous dynamics for each location is governed by an ordinary differential equation (ODE);
- each guard condition is a linear inequality (a hyperplane guard);
- each reset condition is an identity;
- for the hybrid automaton to remain in any location, all guard conditions must be false. This restriction implies that the invariant condition for any location is the convex polyhedron defined by conjunction of the complements of the guards. This is the origin of the name polyhedral-invariant hybrid automaton.

These restrictions are needed to simplify the formal verification task and to allow the simulation of the hybrid system in Simulink/Stateflow, but they certainly reduce the application range.

### 9.3 Building a CheckMate Model

CheckMate models are built with the Simulink GUI using two customized Simulink blocks along with several of Simulink standard blocks. To build the model from scratch, the user must enter the command \texttt{cmnew} at the Matlab command prompt. This will open the CheckMate library from which the user can construct the system model. Currently, the set of blocks used in CheckMate are:
9.3 Building a CheckMate Model

1. Switched Continuous System Block (SCSB). The custom SCSB represents a continuous dynamic system with state equation \( \dot{x} = f(x, \sigma) \), where \( \sigma \) is a discrete-valued input vector to the SCSB and the continuous state vector \( x \) is the block’s output. Currently, three types of dynamics can be specified in an SCSB for each value of the input vector \( \sigma \): clock dynamics \( x = c \), where \( c \) is a constant vector, linear dynamics \( \dot{x} = Ax + b \), where \( A \) is a constant matrix and \( b \) is a constant vector), and nonlinear dynamics \( \dot{x} = f(x) \). The switching function is an \( m \)-file that provides the information about the dynamics of the system. The variable \( \sigma \) selects which dynamics should be used.

2. Polyhedral Threshold Block (PTHB). The other custom block in CheckMate is the PTHB, which represents a polyhedral region \( Cx \leq d \) in the continuous space of the continuous-valued input vector \( x \). The PTHB output is a binary signal indicating whether \( x \) is inside the region or not, i.e. whether or not the condition \( Cx \leq d \) is true. The initial condition, the analysis region and the internal region hyperplane are defined as \texttt{linearcon} object.

3. Finite State Machine Block (FSMB). Discrete dynamics are modeled using a FSMB. FSMBs are regular \texttt{STATEFLOW} blocks that conform to the following restrictions:

- no hierarchy is allowed in the \texttt{STATEFLOW} diagram;
- data inputs must be Boolean functions of PTHB and FSMB outputs only;
- event inputs must be Boolean functions of PTHB outputs only, i.e. events can only be generated by the continuous trajectory leaving or entering the polyhedral regions;
- only one data output is allowed;
- every state in the \texttt{STATEFLOW} diagram is required to have an
entry action that sets the data output to a unique value for that state;

- no action other than the entry action discussed above is allowed in the Stateflow diagram.

Some of these restrictions are rather severe from an ease of use point of view. For example, hierarchy is a much used feature of Stateflow and barring its use may force the designer to enter an unwieldy number of states. Event inputs are in general used to represent disturbances as well as control. Restricting events to represent jumps due to the evolution of the continuous state may again create inconveniences to the user. The other restrictions are made to guarantee deterministic execution of the hybrid automaton.

There are some parameters the user must enter in order to give CheckMate all the details about the verification process. These parameters, as well as any variables used in the Simulink/Stateflow front-end model, are defined and stored in the MATLAB workspace. Parameters and variables can be defined manually or through the use of MATLAB m-files.

9.4 Formal Verification in CheckMate

The verification portion of CheckMate is based on the theory of quotient transition systems [96]. A quotient transition system (QTS) is a finite state transition system that is a conservative approximation of the hybrid system. A QTS is defined from a partition of the state space of the hybrid system with each state in the QTS corresponding to a member of the partition. In the QTS, a transition is defined from a member of the partition $P_1$ corresponding to a set of states in the original hybrid system to another state $P_2$, corresponding to another set of states in the original hybrid system, if and only if there is a state $p_2$ in $P_2$ that is reachable from a state $p_1$ in $P_1$ in
the original hybrid system. A QTS is a conservative approximation in the sense that for every trajectory in the original hybrid system, there is a trajectory in the QTS corresponding to the set of states that the trajectory in the hybrid system goes through. Thus, if we can verify that all trajectories in the QTS satisfy some property, we can conclude that all trajectories in the hybrid system also satisfy the same property. \textsc{CheckMate} only pays attention to the behavior of the hybrid system at the switching instants. Thus, \textsc{CheckMate} approximates the QTS for the hybrid system from the partition of the switching surfaces, which are the boundaries of the location invariants in the PIHA, and the set of initial continuous states.

The transitions in the QTS are based on reachability analysis and, hence, require a very expensive computation for continuous-time dynamical systems. To reduce the computational complexity, reachability analysis is not performed on the original system, but using an approximation method called \textit{flow-pipe approximation} \cite{40}. The flow-pipe approximation is used to define transitions in the quotient transition system for the PIHA as follows. A state in the quotient transition system is a triple $(\pi, p, q)$ where $\pi$ is a polytope in location $(p, q)$ of the PIHA. For each state in the quotient transition system, the flow pipe is computed for the associated polytope under the associated continuous dynamics. The mapping set, the set of states on the invariant boundary that can be reached from $\pi$ is computed. A transition is then defined from $(\pi, p, q)$ to any other state whose polytope overlaps with the mapping from $\pi$. \textsc{CheckMate} then performs model checking on this transition system to obtain a verification result for the desired specification. If the verification returns a positive result, then the program informs the user and terminates. If a negative result is returned, the user is informed and given the option of quitting or allowing \textsc{CheckMate} to refine the approximation and repeating the verification. This process continues until a positive verification result is obtained, or the user decides to quit.
9.5 Discussion

CheckMate has several interesting aspects. First of all, it uses a very popular tool suite to capture the design specifications and to simulate the system. Second, it uses a particular restriction of the general hybrid system model presented in Chapter 2 that allows one to carry out formal verification. Third, it uses conservative approximations to reduce the computation costs of formal verification for hybrid systems.

From a practical viewpoint, the approximation scheme yields computational problems that remain prohibitive when the number of variables is more than 5 to 10 due to the high cost of reachability analysis.

CheckMate represents a most interesting approach to hybrid system verification and design. On the other hand, its model is rather restrictive and it is unlikely it could be used as an interchange format. However, the interchange format to be proposed should be general enough to capture all the aspects of the CheckMate model.
Chapter 10

Masaccio

10.1 The Masaccio Language

Hybrid system described in Masaccio result from a hierarchical specification made of components \cite{78, 79}. As we have already seen in the previous chapters, the concept of hierarchy for the specification of complex systems is quite consolidated. We now point our attention on the possible ways of nesting components in hierarchical systems, since-as we will see below-Masaccio offers the greatest flexibility in this sense.

We have already mentioned the concurrent and sequential hierarchies of some modeling tools like Statecharts \cite{76}, UML \cite{31} and Ptolemy \cite{44}; moreover languages like Shift \cite{47} and Charon \cite{9, 4} address specifically the issue of hierarchical modeling for hybrid systems. However all these modeling formalisms focus on simulation rather than formal analysis \footnote{Actually, as we have seen in the previous chapter, some latest results allow some kind of formal analysis also in Charon.}.

Tools that support compositional semantics analysis are some variants of Statecharts, hierarchic modules and hybrid I/O automata. Statecharts has been extended in \cite{134} with variants that allow compositional semantics analysis, but still suffers of some major limitations, most notably
the absence of support for assume-guarantee reasoning. Hierarchic modules [6] provide both serial and parallel composition and support assume-guarantee, but components can be only discrete, thus there is no way of characterizing a continuous-time behavior. On the other hand hybrid I/O automata [101] can also model continuous-time components but serial composition is not supported.

MASACCIO seems to go a step further, since it is aimed at providing an efficient framework for compositional semantics analysis of hybrid systems. Hence it supports both discrete and continuous time components; moreover such components can be arbitrarily nested and composed via both parallel and serial operators. Finally MASACCIO also offers support for assume-guarantee reasoning, and an illuminating example of such a feature is provided in [79].

10.2 The Syntax

Hybrid systems in MASACCIO are built out of components; these are defined in terms of interfaces, which describe their syntactic structure and executions, which define their semantics.

The interface of a component $A$ consists of:

- A finite set $V^i_A$ of input variables.
- A finite set $V^o_A$ of output variables. The following condition must hold: $V^i_A \cap V^o_A = \emptyset$. The state of component $A$ is an assignment of values to the set of variables $V_A = V^i_A \cup V^o_A$. All variables in MASACCIO are typed, so assignment must be consistent with variable types. The set of all possible state assignments to the variables in $V_A$ will be denoted by $[V_A]$.
- A dependency relation $\prec \subseteq V^i_A \times V^o_A$ between input/output variables.

The meaning of the dependency relation is the following: assume $x \prec$
10.3 The Semantics

Then the value of $y$ can depend without delay on the value of $x$. Specifically:

- for jumps: the value of $y$ after the discrete transition takes place, can depend on the value of $x$ also after the jump.
- for flows: the value of the derivative $\dot{y}$ can depend instantaneously on the value of $\dot{x}$.

MASACCIO requires that the dependency relation be acyclic to guarantee the existence of input/output values (for jumps) or curves (for flows). This condition may seem too restrictive, since input/output values or curves can exist also if some cyclic dependency exists, but has the obvious advantage of avoiding expensive fixed-point calculations. This eliminates some of the potential sources of non determinism in the behavior of the hybrid systems.

- A finite set $L_A$ of interface locations. Locations are point through which the control flow enters/exits the component. The interface specifies for each location $a \in L_A$ a jump entry condition $\Psi^\text{jump}_A(a)$ and a flow entry condition $\Psi^\text{flow}_A(a)$. The component can be entered through a given (jump or flow) location if the corresponding entry condition is satisfied by the current I/O state. Control can exit the component at any location. Typically, exit points are locations with unsatisfiable entry conditions. Therefore, we see that, unlike CHARON, which separates entry from exit locations, in MASACCIO there is no syntactical distinction between entry and exit points of the component.

10.3 The Semantics

The semantics in MASACCIO is specified in terms of behaviors of single components. Given a generic component $A$ its behavior is defined by a set $E_A$ of finite executions. Infinite executions in finite time, i.e., Zeno behaviors, are
not allowed in MASACCIO. Zeno behaviors have been thoroughly addressed in [77], and conditions are available on the hybrid system that prevents Zeno behavior. The user should verify whether one of these conditions apply for the description that he/she enters into MASACCIO.

An **execution** is a tuple of the form:

- \((a, w, b)\)
- \((a, w)\)

where \(a \in L_A\) is an entry location, \(b \in L_A\) is an exit location and \(w\) is a sequence of execution steps, i.e. either flows or jumps. The location \(a\) is called the *origin* of the execution, \(b\) (if present) is the *destination*, while \(w\) is called the *trace*.

A **jump** is a pair \((p, q)\) of I/O states; state \(p\) is called the *source* of the jump, while \(q\) is called the sink. A **flow** is a pair \((\delta, f)\), where \(\delta\) is a non-negative number and \(f : \mathbb{R} \rightarrow [V_A]\) is a function differentiable on the closed interval \([0, \delta]\). The quantity \(\delta\) is called the *duration* of the flow, while \(f(0)\) is the source and \(f(\delta)\) is the sink. Intuitively \(f(t)\) with \(t \in [0, \delta]\), describes the state trajectory for the whole duration of the flow.

Due to consistency reasons, the following property must hold: for all subsequent executions of the system (either jumps or flows), the sink of the first must be the source of the second.

The semantics of MASACCIO is made complete with the interpretation of jump or flow *entry conditions*, as explained in the previous section. We recall that executions can start only if the corresponding entry condition is satisfied, and they terminate when there is no entry condition which can be anymore satisfied.

To prevent a *blocking* behavior of the system, some peculiar properties must be guaranteed. Indeed, in MASACCIO the *input-permissiveness* property holds, which states that any component cannot deadlock no matter how the environment changes the inputs, by either jumping or flowing [79]. Input-permissiveness in MASACCIO is proved in [78].
10.4 Component Operators

Atomic components (discrete or continuous) are those containing only one origin and destination. Traces can only be single jumps for discrete components or single flows for continuous components.

For discrete components: allowed jumps are defined in terms of a \textit{jump predicate}, which constrains the values of I/O states before and after a jump. Usually, such constraints are expressed in terms of difference equations.

For continuous components: allowed flows are determined by a \textit{flow predicate}, usually defined by differential equations on I/O signals; clearly the causality property must hold, that is if \( u \) is the vector of input signals and \( y \) is the vector of outputs, then \( u \prec y \).

Generic components are defined by nested compositions of atomic components. MASACCI0 supports two basic compositional operators: parallel composition and serial composition.

Parallel Composition. Given two components \( A \) and \( B \) their parallel composition is denoted by \( A \parallel B \). The corresponding execution starts at a common location \( L_A \cap L_B \), and it is synchronous for both component, that every jump in \( A \) takes place at exactly at the same time of a corresponding jump in \( B \), and similarly flows in \( A \) are matched by flows in \( B \) having the same duration. When parallel composing the two components, outputs of \( A \) can be inputs of \( B \) and vice versa. MASACCI0 supports preemption: when one of the two components reaches an exit location, the execution of the other component is halted and the control flow exits from \( A \parallel B \).

Serial Composition. Serial composition represents sequencing of behaviors. Given \( A \) and \( B \), their serial composition is denoted by \( A + B \). Executions of \( A + B \) are either executions of \( A \) or \( B \). The set of control
flow locations is the union of those of the two individual components, i.e.
\( L_{A+B} = L_A \cup L_B \). Also the set of variables is the union of the sub-
components’ variables: \( V_{A+B} = V_A \cup V_B \).

The triple \((a, w, b)\) is an execution of \( A + B \) iff either \((a, w[A], b)\) is an
execution of \( A \) or \((a, w[B], b)\) is an execution of \( B \).

**Variable Renaming and Hiding.** It is possible in **Masaccio** to re-
assign names to variables to enable sharing of information among the differ-
ent components. Variables having the same names refer to the same signal.

**Masaccio** also supports variable hiding and location hiding to provide
the language with the property of *encapsulation* (see Chapter 1). However,
in order to prevent deadlocks, locations can be hidden only if their corre-
sponding entry condition is satisfied to allow the control flow to never halt
at those locations.

Hidden variables have local scope, meaning that their values are re-
initialized each time the control flow enters into the corresponding compo-
nent.

### 10.5 Assume-Guarantee in **Masaccio**

**Refinement.** Intuitively, if component \( A \) refines component \( B \), we can
think of \( A \) as being “more specific” than \( B \); from the point of view of
observational semantics, all the traces of \( A \) are also traces of \( B \) (the converse
is in general not true). From an operational point of view component \( A \) may
result from \( B \) by adding some constraints on it, e.g. \( A = B \| C \) for some
other component \( C \).

Formally, refinement in **Masaccio** is defined as follows: component \( A \)
refines component \( B \) if the following conditions are satisfied:

1. every input (output) variable of \( A \) is an input (output) variable of
   \( B \) and the dependency relation of \( B \) is a subset of the dependency
   relation of \( A \). In symbols: \( \prec_B \subseteq \prec_A \).
10.5 Assume-Guarantee in Masaccio

2. every execution of \((a, w, b) \in E_A\) is such that \((a, w[B], b) \in E_B\), that is every execution of \(A\) is an execution of \(B\) provided traces are restricted to only variables belonging to \(B\).

Compositionality. The following property holds in Masaccio: if \(A\) refines \(B\) then \(A||C\) refines \(B||C\) (for a generic component \(C\)), \(A + C\) refines \(B + C\) and also application of variable renaming and variable/location hiding operators do not alter the refinement relation between components \(A\) and \(B\). Moreover, such a result also holds for arbitrary nesting of components via the Masaccio's operators.

Assume-Guarantee. Under some assumptions on the scope variables Masaccio supports the assume-guarantee principle, meaning intuitively that the formal correctness of component \(A\) can be verified assuming its external environment behaves correctly. Such a verification process have to be extended to all the components, and if the formal correctness is proved for each of them, then it follows that the whole system behaves correctly (for details see \([79]\)).
Chapter 11

Shift

SHIFT is a modeling language developed at UC Berkeley for the description of networks of hybrid automata [47, 48, 121]. The name SHIFT is a permutation of HSTIF: Hybrid Systems Tool Interchange Format.

The main difference with the other modeling paradigms is that the overall hybrid system in SHIFT has a dynamically changing structure. More precisely, the whole system in SHIFT is called the world and it consists of a certain number of hybrid components that can be destroyed or created at real-time as the system evolves.

Therefore the SHIFT language is mainly used for the description and simulation of highly complex hybrid systems whose configuration varies over time. The conception of SHIFT was mainly motivated by the specification and analysis of designs for the automatic control of vehicles and highway systems (AHS) [14, 45, 46, 66, 136]. The research area involved is quite rich in issues, going from the design and validation of communication protocols [63, 84] to the verification of safe design [49, 73, 95, 114], and also including the development of suitable implementation methodologies [58, 64].

Hence the need of a modeling framework which is general enough to capture all these distinct issues, while staying at a low level of complexity to facilitate learning and formal analysis. SHIFT was indeed developed to
respond adequately to all these needs.

At the time Shift was developed, other modeling paradigms for the composition of multiple concurrent agents included extended FSM [88], Communicating Sequential Processes [82], DEVS[83], SDL [87] and also the models of computation described in [86, 105, 141]. However none of them had the feature to model dynamic configurations of hybrid components. The characteristic of being able to describe dynamic networks of hybrid systems makes Shift quite unique as a modeling and simulation tool. Areas of application possibly include, together with the mentioned AHS, air traffic control systems, robotics shop-floors, coordinated robotic agents with military applications, like the Unmanned Aerial Vehicles (UAV) (see [89, 90, 127] and the references contained therein).

11.1 The Shift language: syntax and semantics at a glance

The Shift formalism supports the description of dynamic networks of hybrid component. This means that such components can be created, interconnected and destroyed over the temporal evolution of the overall system. Clearly, some communication mechanisms must be provided by the language in order for the hybrid components to exchange information about their status and the external environment. A powerful feature of Shift is that the interaction network itself may evolve.

11.1.1 The world

The whole network of hybrid components is referred to as the world. More formally the world in Shift is a set:

$$ W = \{h_1, \cdots, h_w\} $$

where $h_i$ is called the $i$-th hybrid component in the world. Each component $h$ is, at a specified time, in a particular configuration $C_h$. Hence the
**configuration** (or discrete state) of the world is given by the tuple:

\[ C_W = (C_{h_1}, \ldots, C_{h_w}) \]

The temporal evolution consists of a sequence of *phases*. During each phase the configuration \( C_W \) remains constant and time flows. The configuration of the world is allowed to change only during the transitions from one phase to another. Hybrid components possess both continuous-time and discrete-event dynamics which depend on the configuration of the world. Components follow a continuous-time evolution during each phase and a discrete-event behavior during phase transitions.

In the **SHIFT** language the hybrid components are organized into different *types*. A type is a tuple: \( H = (q, x, C, L, F, T) \), where:

- \( q \in Q \) is the discrete state variable;
- \( x \in \mathbb{R}^n \) is the continuous state variable;
- \( C = (C_0, \ldots, C_m) \) with each \( C_i \subset W \) is the configuration state variable;
- \( L = \{l_1, \ldots, l_p\} \) are the event labels;
- \( F = \{F_q | q \in Q\} \) are the flows;
- \( T \) are the transition prototypes.

### 11.1.2 Types

The type represents the template for the instantiation of actual components. It can also be viewed as a hybrid automaton \( A_H \) having \( Q \) as the set of discrete states. In each state \( q \in Q \) the continuous state \( x \) follows a continuous evolution determined by the flow \( F_q \), which can be of the form of a differential constraint or even a simple algebraic definition. More formally the continuous state \( x \) can be constrained in one of the following ways:
11.1 The Shift language: syntax and semantics at a glance

\( a \) \( \dot{x}_i = F^D_{i,q}(x, x_{C_0}) \) for differential constraints;

\( b \) \( x_i = F^A_{i,q}(x, x_{C_0}) \) for algebraic constraints;

where \( x_{C_0} \) is a vector containing the continuous state variables of all the elements of \( C_0 \).

The set of transitions \( T \) is a set of tuples \( \delta \) of the form:

\[
\delta = (q, q', g, E, a)
\]

where:

- \( q, q' \in Q \) are the source and sink (discrete) states of the transition.
- \( E \) is a set of event labels whose purpose is to synchronize the current component with the rest of the world. An “internal” transition, i.e. a transition which does not synchronize with transitions occurring in the other components of the world, is specified in Shift by leaving \( E \) empty.
- \( a \) is an action that modifies the state of the world. An action may also create or destroy new components.
- \( g \) is a guard condition: it takes the form of a (possibly quantified) boolean expression. It can assume one of two forms:
  1. \( g(x, x_{C_0}) \) (boolean predicate);
  2. \( \exists c \in C_i : g(x, x_{C_0}), 1 \leq i \leq m \) (boolean predicate).

11.1.3 Time and components

Recall from Chapter 2 that the time basis gives a formal definition of time for hybrid systems. The time basis is a sequence of finite or infinite closed intervals over the set of positive reals. A component \( h \) is an actual instantiation of the type \( H \); i.e., it is a map of the kind:

\[
\tau \rightarrow (Q, \mathbb{R}, W^m)
\]
that assigns values to its state variables \((q, x, C)\).

It is always possible to associate to \(h\) a hybrid automaton \(A_h\) derived from the prototypical automaton \(A_H\) using the actual values of \(C\), while the set of discrete states of \(A_h\) is the same of those of \(A_H\) and is clearly equal to \(Q\).

The transitions \(T_h\) of \(A_h\) are derived from those of the prototypical automaton \(A_H\) by applying some specified restrictions, consistent with the Shift run-time system (see the references at the end of this chapter).

### 11.1.4 The world automaton

The whole world in Shift can also be considered as a hybrid automaton \(A_W\). In this case the set of discrete states is given by the cross product \(Q_W = Q_{h_1} \times \cdots \times Q_{h_w}\); similarly the continuous domain of \(A_W\) is the cross product of the sets of continuous states of all the components in \(W\) at a specified time (as the components can be created or destroyed at run-time).

The continuous dynamics are specified by the flows defined in each component automata. The set of transitions \(T_W\) of \(A_W\) are tuples:

\[
\Delta = (\delta_1, \cdots, \delta_w)
\]

where: \(\delta_i \in T_{h_i}\).

The action associated to \(\Delta\) is the parallel execution of the actions of the component transitions. Shift performs synchronous compositions of multiple component automata. A different, alternative way of specifying the compositions of multiple automata can be that of describing pair-wise cause-effect relationships between the transitions of different components [65, 94].

The choice of one of these two approaches basically relies upon a trade-off between efficiency of implementation and ease of use. Models specified using the synchronous approach have a spatial complexity (meant as a measure of allocation of memory needed) which is exponentially smaller than those specified using the cause-effect approach. However, the algorithm for
determining world transitions in the synchronous approach has a computational complexity (meant as a measure of basic computational operations performed) which is exponentially greater than that required by the corresponding algorithm for the cause-effect approach.

Thus, in SHIFT the ease of use has been preferred over the efficiency of implementation.

11.2 Discussion

SHIFT is a modeling paradigm for the description of dynamic networks of hybrid components. The major distinction with respect to other modeling languages for hybrid systems (like CHARON, or MASACCIO) is that in SHIFT the configuration of the examined system (called world in the SHIFT jargon) is dynamic, meaning that it results from the continuous creation/destruction of objects each modeling a distinct hybrid sub-system.

In principle such a dynamics description of networks of hybrid automata can also be carried out using other modeling languages, but it would require additional efforts, since languages like CHARON or MASACCIO are oriented towards a static description of the modeled system, while it comes quite naturally in SHIFT.

SHIFT is both a programming language and a run-time environment for the simulation of dynamic networks of hybrid automata [123]. Also available is a compiler for translating a SHIFT program to a C program.

More recently a new language has been developed by the research group that created SHIFT. Its name is λ-SHIFT [124]. Like its predecessor λ-SHIFT is a specification language for the dynamic networks of hybrid components and it is designed to provide a tool to simulate, verify and generate real-time code for distributed control systems arising in applications like AHIS and the other mentioned before. What really distinguishes λ-SHIFT from its predecessor is the syntax: λ-SHIFT is an extension of the Common Lisp Object System (CLOS) [30, 126]. In particular, in order to
provide a better use of the CLOS capabilities, the Meta-Object Protocol (MOP) [29] has been extended to provide an open and specializable implementation of the λ-SHIFT specification language.
Chapter 12

Hysdel

HYSDEL is a hybrid systems description language publicly distributed by the Automatic Control Laboratory of the Swiss Federal Institute of Technology Zurich. It allows users to describe *discrete hybrid automata* (DHA) and to generate the correspondent piecewise affine system (PWA). In fact, HYSDEL is limited to a subclass of hybrid systems, i.e. affine systems. Also, HYSDEL only consider discrete dynamics.

12.1 HYSDEL Syntax

A HYSDEL netlist has the following structure:

```plaintext
SYSTEM <name> { 
  /* C-style comments */
  INTERFACE {
  
  }
  IMPLEMENTATION {
  
  }
}
```

The interface section describes the following properties of a system:
• **STATE, INPUT, OUTPUT**: respectively state variables, inputs and outputs subsections. State, input and output variables are declared by the type specifier which can be **REAL** for real valued variables, or **BOOL** for boolean valued variables. A type specifier is then followed by the variable name. For real variable an optional interval can be specified by using the suffix \([\text{min}, \text{max}]\) to denote respectively the minimum and maximum value that the variable can assume.

• **PARAMETER**: parameter subsection. A parameter can be specified in one of the following way:

  – **BOOL** name=value; where value can be **TRUE** or **FALSE**.
  – **REAL** name=value; where value is a real number.
  – **REAL** name; in this case the parameter is treated symbolically.

In the sequel we shall indicate boolean signals with a \(b\) subscript and real signals with an \(r\) subscript.

In the implementation section the user describes the behavior of the hybrid system. A system is described using mainly the following sections:

**CONTINUOUS** : continuous sections can be used to describe continuous dynamics. This section contains equations of type \(\text{var} = \text{affine-expression};\). So, the syntax only allows specification of linear systems. Also (as we will see in Section 12.2), \(\text{var}\) is a discrete time variable. Basically continuous sections describe an affine discrete time dynamical system.

**AD** : AD sections are used to define boolean variables from continuous ones. An AD section has a set of statements of type \(\text{var} = \text{affine-expression} \leq \text{real-number} \text{ or } \text{var} = \text{affine-expression} \geq \text{real-number}\). This section can be seen as an analog to digital converter.
12.1 **Hysdel Syntax**

**DA** : DA sections generate continuous variables from boolean ones. The syntax for an item in this section is as follows:

\[ \text{var} = \text{IF Boolean-expr THEN affine-expr} ; \]
\[ \text{var} = \text{IF Boolean-expr THEN affine-expr ELSE affine-expr}. \]

A variable is assigned to an affine expression depending on the value of the boolean expression. Sampling could be one example of DA section where the boolean expression is a clock signal.

**AUTOMATA** : AUTOMATA sections specify the state transition equations of the discrete automata of the hybrid system. This section contains expression of the type \( \text{var} = \text{Boolean-expression} \). A boolean expression can use logic operators like \& (AND), | (OR) and \text{(NOT)}.

**OUTPUT** : OUTPUT sections contain static linear and logic relations and are used to define the output functions of the hybrid system.

The section above are the most important to model hybrid systems in Hysdel. Additionally, the following sections can be specified:

**LOGIC** : these sections are used to define boolean variable with expression like \( \text{var} = \text{Boolean-expression} \). They can be used to define internal boolean variables.

**LINEAR** : these sections are used to define real valued variables. They can be used to define algebraic expressions of real valued variables.

**MUST** : these sections are used to describe constraints on continuous and Boolean variables. Expressions in this section are of the form:

\[ \text{affine-expression} \geq \text{affine-expression} \]
\[ \text{affine-expression} \leq \text{affine-expression} \]
\[ \text{Boolean-expression} \]
12.2 HYSDEL Semantics

HYSDEL systems semantics is defined in terms of *Discrete Hybrid Automata* (DHA) (see Figure 12.1).

![Figure 12.1: Block diagram of a discrete hybrid automata.](image)

The SAS block contains a set of discrete affine systems characterized by the following set of equations:

\[
x_r'(k) = A_{i(k)}x_r(k) + B_{i(k)}u_r(k) + f_{i(k)} \quad \text{(12.1)}
\]

\[
y_r(k) = C_{i(k)}x_r(k) + C_{i(k)}u_r(k) + g_{i(k)} \quad \text{(12.2)}
\]

where \(x_r'(k) = x_r(k+1), x_r \subseteq \mathbb{R}^{n_r}\) is the continuous state vector, \(u_r \subseteq \mathbb{R}^{m_r}\) is the external input, \(y_r \subseteq \mathbb{R}^{p_r}\) is the continuous output vector and, finally, \(\{A_{i(k)}, B_{i(k)}, C_{i(k)}, D_{i(k)}\}_{i \in \mathcal{I}}\) are matrices of suited dimension. \(i(k)\) selects a different set of matrices, and hence a different affine system, depending on \(k\). This means that \(i(k)\) represents a mode of operation. Each mode is
characterized by different discrete dynamics. The mode is computed by a logic function of the boolean state and input. The finite state machine represents the hybrid automata whose state transition depends on the external boolean input $u_b$, the previous state and the boolean variable $\delta_e(k)$,

\[
x'_b(k) = f_B(x_b(k), u_b(k), \delta_e(k)) \tag{12.3}
\]

\[
y'_b(k) = g_B(x_b(k), u_b(k), \delta_e(k)) \tag{12.4}
\]

$\delta_e(k)$ is true when some particular conditions on the continuous variables is satisfied. In particular we have:

$$\delta_e(k) = f_H(x_r(k), u_r(k), k)$$

Basically we have that $\delta_e^1(k) = 1 \iff k T_s \geq t_0$ (where $T_s$ is the sampling time), or $\delta_e^2(k) = 1 \iff a^T x_r(k) + b^T u_r(k) \leq c$. The mode selector is a logic function $i(k) = f_M(x_b(k), u_b(k), \delta_e(k))$.

Reset maps can be considered in this setting as a special dynamics acting for one sampling step. During this step, variables are set to a specific value.

A HYSDEL program has a natural interpretation as a DHA. CONTINUOUS sections can be used to describe affine systems in the SAS block. AD sections can be used to generate $\delta_e(k)$ while DA sections can be used to switch among several affine systems depending on the value of some boolean variables. Finally, the AUTOMATA section is used to describe the finite state machine.

The use of LINEAR sections could lead to algebraic loops. Algebraic loops are statically detected and reported by the HYSDEL compiler. Notice that the discrete nature of an HYSDEL program makes it impossible to describe Zeno automata.

### 12.3 Examples

The first example we show is the classical bouncing ball example. The first step is the declaration of the interface:
SYSTEM bouncingball {
  INTERFACE {
    STATE {
      REAL x [-10,10];
      REAL v[-100,100];
      BOOL down;
      BOOL resetdown; }
    INPUT {
      REAL tt [0.0,20.0];
    }
    PARAMETER {
      REAL T = 0.001;
      REAL g = 9.81; }
  } /* end interface */
  IMPLEMENTATION {...}
}

There are two state variables: \( x \) represents the position and \( v \) the velocity of the ball. For each real variables, the HYSDEL compiler tries to figure out its bounds starting from the variables it depends on. If no bounds are specified for any variable then an error message is returned saying that the bounds cannot be computed. In this example explicit bounds are given to the state variables. There are also two boolean state variables representing the states of the system. As reported in Section 1.2f of the HYSDEL manual [132], a reset map can be written as a new state where the equations are simply the reset conditions. Then, it is possible to define two states corresponding to two discrete affine dynamical systems. The hybrid automata is in state \texttt{down} if the ball is falling, while it goes in state \texttt{resetdown} when the ball hits the ground. Finally, as the ball is going up, the automata is in neither of the
12.3 Examples

two states. There are two parameters and $T$ is the sampling time. Since HYSDEL only deals with discrete dynamics, the user has to discretize the system of equations as part of its modeling effort and provide the sampling time. The Implementation section is described as follows:

IMPLEMENTATION {
  AUX {
    BOOL hit, falling;
    REAL vfree, xfree, xresetdown, vresetdown; }
  CONTINUOUS {
    v = vfree + vresetdown;
    x = xfree + xresetdown;}
  AUTOMATA {
    down = falling & ~hit;
    resetdown = down & hit;}
  AD{
    hit = x <= 0;
    falling = v <= 0;}
  DA{
    vresetdown = {IF resetdown THEN -0.8*v
    ELSE 0};
    vfree = {IF ~resetdown THEN v - g*T
    ELSE 0};
    xresetdown = {IF resetdown THEN x
    ELSE 0};
    xfree = {IF ~resetdown THEN x + v*T
    ELSE 0};}
} /* end implementation */

The AUTOMATA section describes the states in which the system can be. If the ball is falling (down state) and it hits the ground, then the new state will be resetdown. There are some auxiliary variables that are used internally to
generate a boolean flag indicating when the ball touches the ground \((x \leq 0)\) and when the ball is falling \((v \leq 0)\). Position \(x\) and velocity \(v\) are the sum of two real-valued auxiliary variables defined in DA section. \(v_{\text{free}}\) is the velocity of the ball when it is moving freely in the space. When the ball hits the ground, the system enters the \text{resetdown} state where \(v_{\text{free}} = 0\) and \(v_{\text{resetdown}} = -0.8 \times v\). It is during this state transition (which lasts one time step) that a reset map occurs. The simulation results for the bouncing ball example are shown in Figure 12.2.

Figure 12.2: Simulation result for the HysDEL bouncing ball model.

The second example is the simple RC switched systems described in Section 4.3.

SYSTEM circuit {
    INTERFACE {
        STATE {
            REAL vc [0,100];
        }
    }
}
12.3 Examples

REAL i;
BOOL charging;
BOOL discharging;

INPUT {
REAL tt [0.0,20.0];
}

PARAMETER {
REAL T = 0.001;
REAL R = 10;
REAL C = 0.01;
}

IMPLEMENTATION {

AUX {
BOOL top,bottom;
REAL dc;
}

CONTINUOUS {
vc = vc + i*T/C;
i = (dc - vc)/R;
}

AUTOMATA {
charging = (charging & ~top) | (discharging & bottom);
discharging = (discharging & ~bottom) | (charging & top);
}

AD{
top = vc >=4;
bottom = vc <=1;}

DA{
dc = {IF charging THEN 5
     ELSE -5};
}

}

The elements of this circuit are connected in series: variable $i$ represents the current through each component while $vc$ is the voltage across the capacitor. Two discrete states are defined: charging and discharging. If the system
is in charging state, the voltage source value is set to 5V in the DA section, otherwise it is set to \(-5V\).

The circuit equations are described in the CONTINUOUS section. State variable \(i\) is defined as the current in the resistor, which is also equal to the current in the capacitor. Voltage across the capacitor is the integral of the current through it (note that the differential equation has been discretized). Simulation result are shown in Figure 12.3.

![Figure 12.3: Simulation result for the HYSDEL circuit model.](image)

After a model is described using the HYSDEL language it can be compiled with the HYSDEL compiler to generate an input file for a MATLAB simulation (it is also possible to generate a mixed logical dynamical description of the same system). The MATLAB simulation file has the following interface:

```matlab
function [xn, d, z, y] = circuit(x, u, params)
```
12.4 Discussion

It simulates one step starting from the initial conditions $x$, with input $u$ and parameters $\text{params}$. It returns the new state $x_n$, the output $y$, and some auxiliary variables used in the internal representation of a DHA.

The HYSDEL toolkit also provides a wrapper function with the following interface:

\[
\text{function } [XX, DD, ZZ, YY] = \text{hybsim}(x0, UU, sys, params, Options)
\]

where $UU$ is an input vector, $x0$ is the initial condition, $sys$ is the MATLAB simulation file. The \text{hybsim} function simulates the system $sys$ for all samples in $UU$. This is the reason why even if a system has no inputs it is necessary to have at least the time-line as input.

12.4 Discussion

HYSDEL is a language for the description of discrete hybrid automata. The language was developed to target the modeling of discrete time, affine dynamical systems. Hence, it is not a candidate for being an interchange format. Moreover there are other important features that are missing from the language. First of all hierarchy: HYSDEL programs are flat, i.e. it is not possible to instantiate subsystems and compose them (not even the syntax supports it). Features like declaration, instantiation, hiding, and object-orientation are also missing. In fact, it’s not possible to declare objects of any sort and then instantiate them to compose a system of more complex object.
Part IV

Semantics
Chapter 13

The LSV Tagged-Signal Model
13.1 The LSV Model

The LSV tagged-signal model (LSV model) proposed by Lee and Sangiovanni-Vincentelli \cite{98} is a formalism for describing aspects of models of computation. It defines a semantic framework within which models of computation can be studied and compared. It is abstract—describing a particular model of computation involves imposing further constraints that make it more concrete.

The fundamental entity in the TSM is an event: a value/tag pair \((v,t)\). Tags are often used to denote temporal behavior. A set of events (an abstract aggregation) is a signal. Processes are relations on signals, expressed as sets of tuples of signals. A particular model of computation is distinguished by the order it imposes on tags and the character of processes in the model. More formally, given a set of values \(V\) and a set of tags \(T\), an event is an element of \(V \times T\). A signal \(s\) is a set of events, and thus is a subset of \(V \times T\).

A functional (or deterministic) signal is a (possibly partial) function from \(T\) to \(V\). The set of all signals is denoted \(S\). A tuple of \(n\) signals is denoted \(s\), and the set of all such tuples is denoted \(S^n\). The empty signal in \(S\) (one with no events) is denoted by \(\lambda\). For any \(s \in S\), \(s \cup \lambda = s\). In some models of computation, the set \(V\) includes a special value \(\bot\), which indicates the absence of a value. It should be noticed that \((\bot, t) \not\in \lambda\), indeed \((\bot, t)\) does not satisfy \((\bot, t) \cup s = s\) for all \(s \in S\).

The issue of time representation has been central to all modeling efforts. While time has a rather well studied representation in physical processes, this is not always the case in specifications of designs. In fact, we argue that representing specifications using physical time equivalents may result in over specifications and as a consequence, less efficient designs. For example, data manipulation operations can often be performed concurrently as long as certain precedence relations are satisfied. The specifications for these systems have to reflect only the precedence relations, thus leaving several options open for embedding the computation in physical processes that will
13.1 The LSV Model

indeed have a global ordering on the computation. If such ordering in time is used for specifying the system, there is limited or no freedom in selecting the embedding of the computation. The tagged-signal model has been primarily developed to clarify the issue of time, concurrency and communication for embedded systems.

13.1.1 Processes

A process $P$ is a subset of the set of all $n$-tuples of signals $S^n$ for some $n$. A particular $s \in S^n$ is said to satisfy the process if $s \in P$. An $s$ that satisfies a process is called a behavior of the process. Thus a process is a set of possible behaviors, or a relation between signals.

In the LSV model framework, different process representations are obtained by selecting an appropriate order relation on the set of tags $T$. In a timed process, $T$ is totally ordered, i.e., there is a binary relation $<$ on members of $T$ such that if $t_1, t_2 \in T$ and $t_1 \neq t_2$, then either $t_1 < t_2$ or $t_2 < t_1$. In an untimed process, $T$ is only partially ordered. For instance, data flows are represented by untimed processes.

Intuitively, a set of processes operate concurrently, and constraints imposed on their signal tags define communication among them. The environment in which the system operates can be modeled with a process as well.

For many (but not all) applications, it is natural to partition the signals associated with a process into inputs and outputs. Intuitively, the process does not determine the values of the inputs, and does determine the values of the outputs. A process with $i$ inputs and $o$ outputs is a subset of $S^i \times S^o$, where $(S^i, S^o)$ is a partition of $S^n$ and $n = i + o$. Thus, a process defines a relation between input signals and output signals. An $s$ can be written $s = (s_1, s_2)$, where $s_1 \in S^i$ is an $i$-tuple of input signals for process $P$ and $s_2 \in S^o$ is an $o$-tuple of output signals for process $P$.

A process $F$ is functional (or deterministic) with respect to an input/output
partition if it is a single-valued, possibly partial, mapping from $S^i$ to $S^o$. That is, if $(s_1, s_2) \in F$ and $(s_1, s_3) \in F$, then $s_2 = s_3$. In this case, we can write $s_2 = F(s_1)$, where $F : S^i \rightarrow S^o$ is a (possibly partial) function. A process is **completely specified** if it is a total function, that is, for all inputs in the input space, there is a unique behavior.

In a *memory-less process* only inputs with a given tag concur to form outputs with the same tag. The notion of state has traditionally been used for processes with memory to simplify their representation and to provide a powerful analysis and synthesis mechanism. We can formalize the notion of state in the LSV model following some of the classical notions of system theory, (e.g., [137, 112]) by considering a process $F$ that is functional with respect to partition $(S^i, S^o)$. Let us assume for the moment that $F$ belongs to a timed process, in which tags are totally ordered. For any tuple of signals $s$, define $s_{>t}$ to be the tuple of the (possibly empty) subset of the events in $s$ with tags greater than $t$. Two input signal tuples $r, s \in S^i$ are in relation $E^F_t$ (denoted $(r^i, s^i) \in E^F_t$) if $r_{>t} = s_{>t}$ implies $F(r)_{>t} = F(s)_{>t}$. This definition intuitively means that process $F$ cannot distinguish between the “histories” of $r$ and $s$ prior to time $t$. Thus, if the inputs are identical after time $t$, then the outputs will also be identical. $E^F_t$ is obviously an equivalence relation, partitioning the set of input signal tuples into equivalence classes for each $t$. We call these equivalence classes the *states* of $F$. Under certain conditions, the notion of state can be generalized to untimed models of computation, where events are tagged with partially ordered tags. It is sufficient, for example, that a set $A$ of tags exist that are totally ordered with respect to every event in the input and output signals of a process. Then similar equivalence classes can be defined for each tag in $A$.

### 13.1.2 Concurrency and Communication

The sequential or combinational behavior just described is related to individual processes, and general systems will typically contain several coordinated
concurrent processes. At the very least, such processes interact with an environment that evolves independently, at its own speed. It is also common to partition the overall model into tasks that also evolve more or less independently, occasionally (or frequently) interacting with one another. This interaction implies a need for coordinated communication.

Communication between processes can be explicit or implicit. Explicit communication implies forcing an order on the events, and this is typically realized by designating a sender process which informs one or more receiver processes about some part of its state. Implicit communication implies the sharing of a common time scale (which forces a common partial order of events), and a common notion of state.

13.2 Basic Time

In classical transformational systems, such as personal computers, the correct result is the primary concern—when it arrives is less important (although whether it arrives is important). By contrast, embedded systems are usually real-time systems, where the time at which a computation takes place is very important. For example, a delay in displaying the result of an Internet search is annoying, a delay in actuating a brake command is fatal.

As mentioned previously, different models of time become different order relations on the set of tags $T$ in the tagged-signal model. Implicit communication generally requires totally ordered tags (timed processes), usually identified with physical time.

The tags in a metric timed process have the notion of a “distance” between them, much like physical time\(^1\).

Two events are synchronous if they have the same tag (the distance between them is 0). Two signals are synchronous if each event in one signal

\(^1\) Formally, there exists a function $d : T \times T \to \mathbb{R}$ mapping pairs of tags to real numbers such that $d(t_1, t_2) \geq 0$ where $d(t_1, t_2) = 0 \iff t_1 = t_2$, $d(t_1, t_2) = d(t_2, t_1)$ and $d(t_1, t_2) + d(t_2, t_3) \geq d(t_1, t_3)$ for any $t_1, t_2, t_3$. 
is synchronous with an event in the other signal and vice versa.

13.3 Treatment of Time in Processes

A synchronous process is one in which every signal in the process is synchronous with every other signal in the process. An asynchronous process is a process in which no two events can have the same tag. Note that the common usage of the term “asynchronous” refers to processes that are non synchronous. Asynchronous processes in our framework are a subset of non synchronous processes. We believe that distinguishing between asynchronous and non synchronous is important. In fact, asynchronous functional processes always have a unique behavior while all other processes may have problems when feedback connections involving events with the same tag are present. If tags are totally ordered, the process is asynchronous interleaved, while if tags are partially ordered, the process is asynchronous concurrent. Note, however, that for asynchronous processes concurrency and interleaving are, to a large extent, interchangeable, since interleaving can be obtained from concurrency by partial order embedding, and concurrency can be reconstructed from interleaving by identifying “untimed causality”.

13.4 Implementing Concurrency and Communication

Concurrency in physical implementations of processes implies a combination of parallelism, which employs physically distinct computational resources, and interleaving, which means sharing of a common physical resource. Mechanisms for achieving interleaving, generally called schedulers, vary widely, ranging from operating systems that manage context switches to fully-static interleaving in which multiple concurrent processes are converted (compiled) into a single process. We focus here on the mechanisms used to manage communication between concurrent processes.
Parallel physical systems naturally share a common notion of time, according to the laws of physics. The time at which an event in one subsystem occurs has a natural ordering relationship with the time at which an event occurs in another subsystem. Physically interleaved systems also share a natural common notion of time: one event happens before another.

A variety of mechanisms for managing the order of events, and hence for communicating information between processes, exists. Using processes to model communication (rather than considering it as “primitives” of the LSV model) makes it easier to compare different MOCs, and also allows one to consider refining these communication processes when going from specification to implementation. Recall that communication in the LSV model is embodied in the event, which is a two-component entity whose value is related to function and whose tag is related to time. That is, communication is implemented by two operations:

1. the transfer of values between processes (function; LSV model event value),
2. the determination of the relationship in time between two processes (time; LSV model event tag).

Unfortunately, often the term “communication” (or data transfer) is used for the former, and the term “synchronization” is used for the latter. We feel, however, that the two are intrinsically connected in embedded systems: both tag and value carry information about a communication. Thus, communication and synchronization, as mentioned before, are terms which cannot really be distinguished in this sense.

13.5 Representing Models of Computation with the LSV Model

The description of the physical system uses an abstraction of time that involves sequences, i.e., what matters is the sequences of events not the
precise time at which these events occur. Timed MOCs (recall that timed MOCs include both models that are based on physical time and those based on sequences of events) are used in the description of a system. In particular, we present the categorization of four models of computation of interest in the control community:

- **Finite State Machines.** An FSM is a synchronous LSV model process in which the tags take values in the set of integers and the sets of inputs, outputs and states are finite. The tags represent the ordering of the sequence of events in the signals, not physical time, and are globally ordered. A Finite Automaton (FA) is an FSM with no outputs;

- **Sequential Systems.** An SS is a synchronous LSV model process in which the tags take values in the set of integers and the inputs, outputs and states assume values on infinite sets;

- **Discrete-Event Systems.** In a DES, tags are order-isomorphic with the natural numbers and assume values on the set of real numbers. The tags represent the values of the time at which the events occur.

- **Continuous–Time Systems.** A CTS is a metric timed LSV model process, where $T$ is a connected set.

### 13.6 Discussion

While the LSV model is very useful to compare models of computation and to provide a general framework where the combination of different models of computation are introduced from specification to implementation can be analyzed with rigor, it does not provide per se a theory on how to combine heterogeneous models.
Chapter 14

Trace Algebras

This chapter is based on a denotational view of hybrid systems worked out through the trace concept \cite{34, 35, 36, 37}. A trace is intuitively a representation of a single instance of the behavior of a generic system\footnote{This holds true not only for hybrid systems, but for every system which superimposes a partial order on the succession of events}.

Under this perspective traces are formally defined as mathematical objects, with some associated operators satisfying a set of axioms and thus defining a trace algebra.

Systems behaviors are thus seen as a collection of traces, also called trace structures. Trace operators can also be extended to trace structures with the consequence of defining an algebra of systems (under a denotational point of view) called trace structure algebra.

The main advantage of this formalism is providing a useful methodology for encompassing, under a single modeling paradigm, systems described through different languages. This unifying feature therefore enables dealing with different kinds of processes (finite state machines, discrete event systems, dynamical systems, and of course hybrid systems, to mention only a few), analyzing their possible compositions and abstracting away all the unnecessary details for the correct specification and verification of the overall
system. Note that this approach has strong similarities to the behavioral approach to system theory as proposed by Jan Willems and co-workers that we discussed in Chapter 15.

In what follows we use the word ‘agent’, ‘process’ and ‘system’ as synonyms denoting generic systems with some computational capabilities and with (possibly) some timing constraints. Hybrid systems can of course be derived as a particular subclass of such systems.

### 14.1 Introduction

The concept of trace is not new in the scientific literature dealing with real-time concurrent systems.\(^2\)

A real-time system is basically any process having constraints not only on the order of its events, but also (and this is the most peculiar characteristic) on the times at which they occur. Concurrency refers to the possibility of having distinct simultaneous signals interacting with each other.

Hoare uses the concept of traces to define a denotational semantics, called *trace semantics* for Communicating Sequential Processes in \([81]\) and \([82]\); here the word trace is used in order to model the behavior of a given process as a finite sequence of events. Processes are modeled as collections of traces, possessing the property of prefix-closure as to enable sequential compositions of distinct agents.

Subsequent extensions (like the possibility of modeling *deadlocks*) to Hoare’s work are described in \([32, 33, 119]\).

The concept of trace was later refined with the explicit introduction of time in agents behaviors, leading to the possibility of modeling true real-time systems. A hierarchy of real-time models making use of the trace formalism was first described in \([115]\) and \([116]\) introducing the concept of timed traces.

\(^2\)The former meaning of trace is however extended in the trace algebra formalism we review here, as to also encompass untimed systems.
A timed trace is basically a pair \((t, a)\) where \(a\) is a given action, while \(t\) is a tag denoting the time at which the action has occurred.

Traces have also been used to denote sequences of transitions in asynchronous circuits in [117, 52, 50]. In [52, 50], in particular, Dill developed a methodology for the automatic verification of speed-independent asynchronous circuits. This approach was further refined in [51] to include infinite traces for the representation of liveness properties (28).

One of the major issues for the analysis of real-time systems is that of concurrency. Traces in the CSP framework define an *interleaving semantics* and do not support concurrency explicitly (since behaviors are simply derived as sequences of distinct events). Simultaneity of actions can be introduced in this setting by modeling traces as a sequence of collections of events (see [23]). Other models of concurrency can be found in the Mazurkiewicz traces [103] and partially ordered multisets (POM) [113]. All these results have also been extended to pure real-time systems (see [10] for an excellent survey on this topic).

### 14.2 The Trace Algebra Approach

In all the models described above there are some common features that can be further investigated in order to collect all the different models under a unitarian framework. This is the spirit of the trace algebras formalism. In this regard a trace is no longer derived from a specified computational model, but it is rather defined as a mathematical object, supporting some abstract operators with respect to which the set of traces is closed.

The aim of such an abstract description is supporting the vast majority of models existing in the literature, to capture and formally define their main properties under a unifying perspective, and to abstract away all the unnecessary details which are not useful to analyze the system’s properties of interest.

*Models of computation (MoCs)* have become in the recent years a stan-
standard way of representing different paradigms for computation, communication, and also real-time issues [54]. Typical recurrent examples include Mealy FSMs, Kahn Process Networks, and Petri Nets. All these MoCs have the advantage of best capturing the key aspects of a process of interest. The major difficulty, however, is enabling a system composed by heterogeneous components, to be modeled, and -more importantly- to be analyzed in a unitarian manner. This is one of critical issues in embedded software research today.

We have seen in Chapter 13, a first attempt to cope with such a problematic issue with the LSV model [98]. In this work several different MoCs are analyzed and compared through a common denotational framework. However, the tagged-signal modeling paradigm while extremely powerful and versatile for modeling purposes, could not yet be used to analyze the properties of a heterogeneous system, since it lacks of any operational characteristic.

The theory of trace algebras, while still in its infancy, goes a step further. It has the same unifying features of the LSV tagged-signal model, but it also includes some basic operations (which will be introduced later on) as to embed in its framework the basic compositionality results of the various Models of Computation.

In this perspective, a suitable tradeoff between two distinct objectives must be carried out: first, one wants to keep the framework as general as possible in order to encompass the vast majority of semantic domains; second, one wishes to provide enough structure as to analyze the basic properties of systems described in different models of computation. The trace algebra approach indeed tries addressing the two distinct issues in the most efficient and functional way: the definition of semantical operators in this setting is quite general and powerful enough to describe and analyze the most peculiar aspects of systems belonging to different semantic domains. Moreover the multiple levels of abstraction methodology worked out through conservative
14.2 The Trace Algebra Approach

approximations (see below), gives the possibility of stepping from one semantic domain to another, while preserving the relationships that enable to assess the correctness of the design.

14.2.1 Traces and Trace Algebra Operators

A trace is a mathematical object that models a single execution of a given agent. However, traces would be generic objects without the definitions of suitable operators on them. We will introduce these operators shortly below, but we want to emphasize the fact that the set of traces along with the related operators defines an algebraic structure called the trace algebra. Indeed, the only requirement is that the set of traces must be closed with respect to the application of the trace operators, and this can be easily seen from the definition of the operators themselves.

Following a denotational approach to systems modeling, agents are thus modeled as a set of traces, which defines the set of all possible behaviors for the considered agent. The set of traces modeling a given process is called trace structure. Operators defined on single traces can also be extended to trace structures, thus defining, in a straightforward manner, a particular trace structure algebra associated to the specified trace algebra.

Before going into more details, we divide the space of traces into two subclasses: a) complete traces, and b) partial traces. Complete traces refers to behaviors that don’t have an endpoint; partial traces, instead, model behaviors with endpoints, and they can be used as prefixes of other traces (either partial or complete).

The operations defined on traces are: projection, renaming and concatenation. The concatenation operator is used to define the notion of prefix of a trace; in fact, we assume that $x$ is a prefix of a trace $z$ if there exists a trace $y$, such that $z$ is equal to $x$ concatenated to $y$.

The specification of a trace algebra to model hybrid systems can be done in several ways, depending on the syntactical domain of interest. The most
complete is that of \texttt{metric time}, although other modeling approaches are also possible. In the metric time semantic domain partial traces are triples $x = (\gamma, \delta, f)$, where $\gamma$ is a set of signals associated to the given agent, $\delta$ is the temporal duration of the behavior modeled by $x$, and $f$ maps a signal $v$ in $\gamma$ to a function of time (and it represents the temporal execution of the agent restricted to the signal $v$).

After having defined hybrid systems traces in the metric time domain, in order to complete the description of the corresponding trace-algebra, we have to define the operations of projection, renaming and concatenation on traces.

- The \textit{projection} operator $\text{proj}(B)(x)$ is a restriction to the signals in $B$ of the agent’s behavior. It is used to model the operation of scoping; internal variables are hidden, but not removed from the model. In this way it increases the non-determinism but not the number of possible behaviors.

- The \textit{renaming} operation is denoted by $x' = \text{rename}(r)(x)$, where $r$ is a bijection function that associates to the tuple of signals in $x$ another tuple of signals. It is used to model the operation of instantiation: it takes an agent and produces an agent with a different set of signal names.

- The definition of the \textit{concatenation} operator is a little more involved: the concatenation $x_3$ of $x_1$ and $x_2$, denoted $x_3 = x_1 \cdot x_2$ is defined if and only if $\gamma_1 = \gamma_2$ and

$$f_1(a)(\delta_1) = f_2(a)(0).$$

\footnote{In the sense that it is the one which provides most information of systems behaviors}
14.2 The Trace Algebra Approach

In that case, \( x_3 = (\gamma_1, \delta_3, f_3) \), where:

\[
\begin{align*}
\delta_3 &= \delta_1 + \delta_2, \\
f_3(a)(\delta) &= f_1(a)(\delta), \quad \text{for: } 0 \leq \delta \leq \delta_1 \\
f_3(a)(\delta) &= f_2(a)(\delta - \delta_1), \quad \text{for: } \delta_1 \leq \delta \leq \delta_3
\end{align*}
\]

**Non-metric Time.** At a higher level of abstraction, hybrid systems can be modeled by means of a trace algebra in which only discrete actions (i.e. jumps) are represented. In this case the concept of flow is abstracted away and there is no notion of time. The sequence of actions is captured by the global order imposed on the discrete events of the system being represented. In this case also, traces can be formally represented, along with the definition of the operators: projection, renaming and concatenation.

**Pre-Post Time.** Going a step further, it is also possible to abstract away the sequence of discrete actions in hybrid systems behaviors, only leaving the initial and final states of the agent’s execution. This is, for example, useful when it is necessary to specify the constructs of a programming language. Here again, a suitable trace algebra can be defined as to capture only the required behaviors of agents.

14.2.2 Homomorphisms and Conservative Approximations

Trace algebras describing systems behaviors at different levels of abstraction can be related by means of conservative approximations. Basically, a conservative approximation is a map of a trace structure in a lower level abstraction domain to another trace structure in a higher level abstraction domain. Let \( \mathcal{A} \) and \( \mathcal{A}' \) the trace structure algebras corresponding to the lower and higher level of abstraction, respectively. For instance, \( \mathcal{A} \) can be the trace structure algebra associated to the metric-time domain, while \( \mathcal{A}' \) can be the trace structure algebra associated to the non-metric time domain.
Trace Algebras

More formally, a conservative approximation is defined by a pair of functions \( \Psi = (\Psi_l, \Psi_u) \), each mapping trace structures in \( \mathcal{A} \) to trace structures in \( \mathcal{A}' \). Let \( T \) be a trace structure (i.e. a set of behaviors) in \( \mathcal{A} \); loosely speaking \( \Psi_u(T) \in \mathcal{A}' \) represents an upper bound of the behaviors described by \( T \), meaning that it includes all the behaviors of \( T \), plus possibly some more, mapped in the higher level abstraction domain \( \mathcal{A}' \). Conversely, \( \Psi_l(T) \in \mathcal{A}' \) is a lower bound of \( T \) (it contains only "abstract" behaviors of \( T \) in the domain \( \mathcal{A}' \), but possibly not all of them).

This has the following, remarkable implication:

\[
\Psi_u(T_1) \subseteq \Psi_l(T_2) \Rightarrow T_1 \subseteq T_2
\]

for any \( T_1, T_2 \in \mathcal{A} \).

This property can be conveniently used when dealing with verification problems. Indeed, assume that \( T_2 \) is a specification for a given process, while \( T_1 \) is an actual implementation. Hence, using the above result, it is possible to check if the implementation satisfies the constraints imposed by \( T_2 \) in the abstract domain \( \mathcal{A}' \), where it is presumably more efficient. It should be observed, however, that the preceding result only enables to discard the occurrence of false positives, while does not rule out the possibility of obtaining false negatives.

Conservative approximations can be derived, in a straightforward way, from a homomorphism between the trace algebras \( \mathcal{C} \) and \( \mathcal{C}' \) associated to the two distinct domains of abstractions. Hence, \( \mathcal{A} \) and \( \mathcal{A}' \) are the trace structure algebras induced by the trace algebras \( \mathcal{C} \) and \( \mathcal{C}' \).

In the case examined here, a homomorphism \( h \) is a function \( h : \mathcal{C} \rightarrow \mathcal{C}' \) that commutes with the operations of projection, renaming and concatenation. Let us denote by \( P \in \mathcal{A} \) a generic agent in the lower level domain of abstraction; it can be shown that the function \( \Psi_u \) for the conservative approximation \( \Psi = (\Psi_l, \Psi_u) \) we want to derive, is given by:

\[
\Psi_u(P) = h(P)
\]
While the function $\Psi_l$ is given by:

$$\Psi_l(P) = h(P) - h(C - P)$$

In conclusion, in order to derive a conservative approximation between two distinct domains of abstraction, it is possible to define the trace algebras associated to the different domains and subsequently find a homomorphism between them. Clearly, such a process can be iterated several times to obtain conservative approximations between domains at multiple levels of abstraction.

14.3 Discussion

Trace algebras have a great degree of generality. They can describe systems specified in many different semantic domains, yet they preserve some basic structural operators that allow a complete (in terms of the design objectives) analysis of the systems considered.

In this brief introduction to this area of research, however, we focused on the use of the trace algebra formalism in order to describe and analyze a particular class of processes: hybrid systems. Due to the complex behavior of this class of system, it would be impossible to reproduce all the aspect of hybrid automata. However, the trace algebra analysis, in conjunction with the multiple levels of abstraction methodology worked out through conservative approximations, enables the formal verification of many several properties imposed on the design specification, thus making it an invaluable tool to support the specification, design and analysis of the most general class of hybrid systems.

It is thanks to these powerful features of the trace algebra formalism, that a direct comparison can also be carried out with respect to other, indeed more complete, modeling paradigms for hybrid systems, like the ones we encountered in the first deliverable.
In MASACCIO, for example, the representation is based on components that communicate with other components by sharing common locations for the control, and variables for data. As for trace algebras, in MASACCIO the behavior of a single component is also given by a sequence of simple executions which can be either jumps (discrete transitions) or flows (continuous execution).

Compositionality in MASACCIO is achieved by means of two distinct operators: parallel composition and serial composition, which have the same meaning of the corresponding operators in the trace algebra framework. Strictly speaking, serial composition in MASACCIO is only the disjunction of two (or possibly more) distinct behaviors: each execution of the composition needs only to satisfy the execution of one of the components. It thus does not provide proper serialization, though. However, by using serial composition in conjunction with the operator of location hiding it is always possible to string together (i.e. serialize) distinct behaviors, in such a way as to offer proper serial composition of behaviors.

The biggest difference between MASACCIO and Trace Algebras however relies on the distinct approach to semantics of the model.

Trace algebras support only a denotational semantics, meaning that the systems can only be described (and analyzed, of course) by the composition of different behaviors, which can be thought of as partial functions of time, specified a priori.

MASACCIO, on the other hand, also supports an operational semantics, in that it provides an implicit description of behaviors by means of suitable constraints (e.g. differential equations, algebraic equations, invariants, guards, etc.), defining on the whole a system made of the composition of several hybrid automata.
Chapter 15

Behavioral Approach

The behavioral approach gives a formal framework for modeling and for the analysis of dynamical systems [133, 137, 138, 139, 140]. Unlike the standard theory, the behavioral point of view does not distinguish between input and output signals, which are both dealt with using a unified approach. This is perhaps the most evident difference with respect to the classical approach, and the starting point for an alternative theory for dynamical systems. A compositional framework for hybrid system specification that is behavioral in nature is proposed by Benveniste in [22] together with a method to automatically synthesize proper scheduling constraints for the joint simulation of discrete-time, event-based, and continuous-time components. Recent attempts to modeling hybrid systems through the behavioral approach can be found in [135] and [91, 92].

The behavioral framework is strictly keen to that of trace algebras, in that it shares a common denotational approach to defining semantics. Also, the interconnection operator is very similar (if not strictly identical) to the parallel composition in trace algebras, and other operators can be easily imported in the behavioral setting from the trace algebra formalism and for this reason, we included this Chapter in the review.
15.1 The Behavioral Approach

Dynamical systems theory investigates the relations between different signals of time, i.e. trajectories; time is seen as an independent variable, while trajectory values are considered as dependent variables. Roughly speaking, the behavior of a system is the set of all its possible trajectories.

Let $T$ be the set of independent variables ¹ and let $W$ be the set of dependent variables, that is the values of system’s trajectories. Here and in the following, with abuse of notation, we will use the symbols $T$ and $W$ to denote either the set of variables for a behavioral system (independent or dependent) or the values they can respectively assume. Each time, the difference between the two meanings will be clear from the context.

The set $W^T$ is called the universum of signals, and it contains all possible behaviors, those compatible with the system at hand and those which are not. A given system has to specify a sort of constraint on the universum $W^T$ in order to retain only behaviors which are possible realizations for the systems itself and rule out the others. More formally, a system has to specify the set $\mathcal{B} \subset W^T$ of its possible behaviors.

The definition of a dynamical system in the behavioral approach is thus given by the triple:

$$\Sigma = (T, W, \mathcal{B})$$

where $\mathcal{B}$ is called the behavior of the system; it defines which signals $w : T \rightarrow W$ are compatible with the model of the systems: those belonging to $\mathcal{B}$ are compatible, while those that do not belong to $\mathcal{B}$ are not.

We observe that in the previous definition the notion of input and output signals is completely absent. From the behavioral point of view, such a dis-

¹In the vast majority of cases $T$ can be identified with time (either continuous or discrete), but there are some important exceptions. For example, in the case of distributed systems both time and space have to be considered as independent variables.
15.1 The Behavioral Approach

tinction is unnecessary and may generate confusion in those cases where the 
input/output partition does not reflect the physical meaning of the model; 
an example, in this sense, could be the distributed system described by the 
Maxwell equations: here we have the electric field, the magnetic field, the 
current densities and the charge densities as dependent variables; however 
it is not clear (nor it has much sense from a physical point of view) which 
of these quantities can be represented either as input or output signals.

Kernel representation of LTI systems. A linear time-invariant dif-
fferential system is a behavioral system $\Sigma = (\mathbb{R}, W, \mathcal{B})$, with $W = \mathbb{R}^m$ a 
finite-dimensional vector space, whose behavior $\mathcal{B}$ consists of the solutions 
of a system of differential equations of the form:

$$R_0 w + R_1 \frac{d}{dt} w + \cdots + R_n \frac{d^n}{dt^n} w = 0$$

where $R_i, (i = 1, \ldots, n)$ are matrices of appropriate size. Using a polynomial 
matrix notation, the system of differential equations above can be rewritten 
as:

$$R \left( \frac{d}{dt} \right) w = 0$$

where $R$ is real polynomial matrix with $m$ columns.

The behavior of this system is thus defined as:

$$\mathcal{B} = \{ w : \mathbb{R} \to \mathbb{R}^m | R \left( \frac{d}{dt} \right) w = 0 \} = \ker(R \left( \frac{d}{dt} \right))$$

and we call it a kernel representation of the LTI system $\Sigma$.

15.1.1 Input/Output Representation

So far we have defined linear differential systems using the behavioral paradigm. 
However, starting from the behavioral definition of such a system, an equiva-
\ alent input/output representation can always be found.
Formally speaking, given a linear differential system $\Sigma = (\mathbb{R}, \mathbb{R}^n, \mathcal{B})$, there exist a permutation matrix $\Pi$ and a partition $\Pi w = (u, y)$ of variables $w$ such that for any smooth function of time $u^*$ (i.e. $u^* \in C^\infty(\mathbb{R}, \mathbb{R}^u)$), there exists a smooth function of time $y$ such that $(u^*, y) \in \Pi \mathcal{B}$. Moreover, $y$ such that $(u^*, y) \in \Pi \mathcal{B}$ forms a linear finite dimensional variety, implying that $y$ is uniquely determined by its derivatives at time $t = 0$.

The implication of this result is that linear differential systems can always be partitioned into input and output variables. Independently of their actual physical meaning, input variables are considered as free variables, while output variables are bound, in the sense they can be completely determined by the inputs and their initial conditions.

15.1.2 Modular Representation of Behavioral Systems

The behavioral systems framework naturally lends itself to model interconnected systems, say systems composed of several (possibly nested) modules.

In such a modeling paradigm, the basic building blocks are the modules, each equipped with a collection of distinct terminals. Terminals carry variables assuming values in a given domain (e.g. boolean, the set of integers, reals, positive integer/reals, etc.). To each variable is associated a corresponding signal in the time domain.

The behavior of modules is modeled by some specified dynamical laws, also determining the behavior of signals at the module’s terminals, which represent the component of the module’s behavior externally visible.

Terminals from distinct modules can be interconnected together, provided they are defined on the same domain of values. Such an interconnection clearly induces some constraints on the value that a variable at both terminals can assume, since the collection of signals at the interconnected terminals is the intersection of signals at the same terminals before they were interconnected.
15.1 The Behavioral Approach

Several distinct modules connected together by means of their terminals define an *interconnection architecture*. The behavior of the interconnected system is defined in terms of the signals that satisfy both the modules behaviors and the interconnection constraints induced by the interconnection architecture.

Formally, a module $M$ with $N$ terminals yields $W = W_1 \times W_2 \times \cdots \times W_N$ as co-domain, where $W_k$ is the co-domain associated to the $k$-th terminal. The behavior of the module is thus the set $\mathcal{B}(M) \subset W^\mathbb{R}$ and the (behavioral) system associated to $M$ is $(\mathbb{R}, W, \mathcal{B}(M))$.\(^2\)

In the formalization of a generic interconnected system, it is convenient to represent as **manifest variables** the variables associated with external terminals and as **latent variables** the internal variables associated with the terminals that are paired by the interconnection architecture. The co-domain of the manifest variables is thus:

$$W = (W_{e_1} \times \cdots \times W_{e_{|E|}})$$

where $E = \{e_1, \ldots, e_{|E|}\}$ is the set of external terminals; the co-domain of the latent variables, instead, is:

$$L = (W_{i_1} \times \cdots \times W_{i_{|I|}})$$

where $I = \{i_1, \ldots, i_{|I|}\}$ is the set of internal terminals.

The full behavior $\mathcal{B}$ of the interconnected system is given by the collection of the single behaviors of the composing modules, subject to the constraints defined by the interconnection architecture. The overall interconnected systems is thus represented by the **latent variables system**:

$$\bar{\Sigma} = (T, W, L, \mathcal{B})$$

\(^2\)This is clearly an “external” modeling perspective of the modular framework, in that it represents the system only in terms of its externally visible variables, i.e. the variables associated to terminals. It does not take into account internal signals of the module.
15.1.3 Controllability and Observability

The notion of controllability in dynamical systems refers to the possibility of transferring the given system from one state to another by means of a suitable control action. Clearly enough, this property might be desirable whenever we want to transfer the system state from an unsafe mode of operation to a safety region.

For state-space systems of the form \( \frac{dx}{dt} = f(x, u, t) \) we say that the system defined by the controlled vector field \( f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^n \) is controllable if \( \forall x_1, x_2 \in \mathbb{R}^n \) there exist an input \( u \in (\mathbb{R}^m)^{\mathbb{R}^+} \) and \( t_F \in \mathbb{R}^+ \) such that the solution of \( \frac{dx}{dt} = f(x, u, t) \) with initial condition \( x(0) = x_1 \) yields \( x(t_F) = x_2 \).

Controllability is related to two distinct factors:

- the choice of the initial state;
- the choice of the control action.

Thus it would be desirable to make a distinction between these two aspects, in order to acquire more generality. In the behavioral setting indeed controllability refers to a more general notion of observability, as the ability to switch from any one trajectory in the behavior to any other one allowing some finite time delay. More formally, given a dynamical system \( \Sigma = (T, W, \mathcal{B}) \) (with either \( T = \mathbb{R} \) or \( T = \mathbb{Z} \)), \( \Sigma \) is said to be controllable if for \( \forall w_1, w_2 \in \mathcal{B} \) there exists \( \tilde{t} \in T, \tilde{t} \geq 0 \) and \( w \in \mathcal{B} \) such that \( w(t) = w_1(t) \) for \( t < 0 \) and \( w(t) = w_2(t - \tilde{t}) \) for \( t \geq \tilde{t} \).

The notion of observability in behavioral systems is somewhat similar to that of controllability. Given a dynamical system \( \Sigma = (T, W, \mathcal{B}) \), and let \( W = W_1 \times W_2 \) for suitable subspaces \( W_1 \) and \( W_2 \). We say that \( w_1 \in W_1 \) is observable from \( w_2 \in W_2 \) in \( \Sigma \) if \( (w_1, w_2') \in \mathcal{B} \) and \( (w_1, w_2'') \in \mathcal{B} \) implies that \( w_2' = w_2'' \). Loosely speaking, a system is observable if it is always possible to deduce \( w_1 \) from the observation of \( w_2 \) and from the dynamical laws of the system (of course the behavior \( \mathcal{B} \) is assumed to be known).
15.2 Hybrid Systems and the Behavioral Approach

The behavioral approach can also serve the purpose of modeling and analyzing hybrid systems. Under this perspective hybrid systems can be dealt with using a pure denotational approach, like it happens for trace algebras. However, a general theory of behavioral hybrid systems has not yet been formulated. The major contributions in this area usually relate to some particular classes of hybrid systems, like the ones we are going to examine below. Nevertheless, we reckon this field of research is very promising, and expect that more general results will be available soon.

15.2.1 Behavioral State Representation of Hybrid Systems

The behavioral framework can be usefully exploited as a formal tool to analyze discrete event systems (DES) and continuous dynamical systems under a unifying perspective. This also allows us to deal with hybrid systems, as they exhibit both the continuous and discrete event features.

This is made possible by introducing a particular class of behavioral systems called state systems. The intuitive meaning of grouping systems in this class is that they have a direct counterpart in terms of finite automata (which is the class of DES we will focus on) and continuous dynamical systems.

Given a latent variables system:

\[ \Sigma_X = (T, W, X, B) \]

we say that \( \Sigma_X \) is a state system if, given two distinct behaviors of the system the concatenation of their “latent behaviors” at time \( t \in T \) is also a behavior of the system. More formally:

\[ \text{And indeed we will see Section 15.3 that behavioral systems and trace algebras formalisms are strictly intertwined} \]
Behavioral Approach

if \((w_1, x_1) \in B\) and \((w_2, x_2) \in B\) and there exists \(t \in T\) such that \(x_1(t) = x_2(t)\) then:

\[ (w_1, x_1) \land_t (w_2, x_2) \in B \]

where \(\land_t\) is the concatenation operator at time \(t\):

\[ (f_- \land_t f_+)(t') := \begin{cases} 
  f_-(t') & \text{for } t' < t \\
  f_+(t') & \text{for } t' \geq t 
\end{cases} \]

If \(\Sigma_X\) is a state system that induces the manifest system \(\Sigma\), then we say that \(\Sigma_X\) is a state representation of \(\Sigma\).

State systems can suitably describe both finite automata and continuous dynamical systems. More importantly they can also describe hybrid systems, thanks to their concatenation property that allows them to model sequencing of discrete event and continuous behaviors, resulting in an overall hybrid behavior.

A finite automaton is a triple \((A, S, \phi)\) where \(A\) is an alphabet of symbols, \(S\) is the set of states, and \(\phi\) is a partial map from \(S \times A\) to \(S\), modeling the transition relation. The intuitive meaning is that if the system is in state \(s \in S\) it accepts the event \(a \in A\) if and only if \((s, a)\) is in the domain of \(\phi\). It is easy to verify that an automaton can be described by a behavioral state system where the latent variables are the states of the automaton and the manifest variables are the symbols accepted by the automaton. Disregarding the behavior at time \(\pm \infty\), the time basis can be set equal to the set of integer numbers \(\mathbb{Z}\). The automaton restricts the possible behaviors to those sequence of state/events belonging to the domain of \(\phi\).

As far as continuous dynamical systems are concerned, the usual state-space representation is the following:

\[
x(t + 1) = f(x(t), u(t)) \\
y(t) = h(x(t), u(t))
\]
for discrete-time systems, and:

\[
\frac{dx}{dt} = f(x(t), u(t)) \\
y(t) = h(x(t), u(t))
\]

for continuous-time systems, where \( x \in \mathbf{X}, u \in \mathbf{U}, y \in \mathbf{Y} \).

The behavioral representation of such models in terms of state-space systems is quite straightforward and is extremely similar to that for finite automata. Indeed we can think of the input/output pair \((u(t), y(t))\) as events, with the assumption that whatever the state the system is in, the input event can be chosen freely and the output event follows from \( h(\cdot, \cdot) \).

Continuous discrete-time systems have a direct representation in terms of behavioral state space models if we let the manifest variables coincide with the input/output pairs, and the latent variables with pairs in \( \mathbf{U} \times \mathbf{Y} \times \mathbf{X} \times \mathbf{X} \) for discrete-time systems and \( \mathbf{U} \times \mathbf{Y} \times \mathbf{X} \times T\mathbf{X} \) for continuous-time systems (where \( T\mathbf{X} \) is the tangent bundle of \( \mathbf{X} \)).

More precisely, the full state behavior for discrete-time systems is described by \((u(t), y(t), x(t), x(t+1))\) \(\in\mathcal{B}_0\), with \( \mathcal{B}_0 \) being a subset of \( \mathbf{U} \times \mathbf{Y} \times \mathbf{X} \times \mathbf{X} \); for (smooth) continuous-time systems the full state behavior is described by \((u(t), y(t), x(t), dx/dt)\) \(\in\mathcal{B}_0\), with \( \mathcal{B}_0 \) being a subset of \( \mathbf{U} \times \mathbf{Y} \times \mathbf{X} \times T\mathbf{X} \).

We observe, that in both cases \( u(t) \) and \( y(t) \) are the manifest variables, and \( x(t), x(t+1) \) (or respectively, \( x(t), dx/dt \) for continuous-time systems) are the state variables.

It is now clear that both finite automata and continuous systems can be modeled and analyzed using the same behavioral paradigm, based upon behavioral state models. Thanks to the concatenation operator of this class of systems it is straightforward to include both discrete event and continuous behavior into the same behavioral state system, thus resulting in an extremely powerful and general way of modeling hybrid systems.
15.2.2 Concatenability of Behaviors in Hybrid Systems

In this section we examine hybrid systems interconnections under a behavioral perspective. Interconnections of hybrid systems usually admit an instantaneous reset of the quantities involved, due to the switching nature of control strategies or external disturbances or even an intrinsic characteristic of the systems itself (*invariance transitions*).

As it is well known, continuity of signals is not generally required for hybrid systems. However, when composing different hybrid modules we have to ensure that the overall system does not exhibit any behaviors which are not physically feasible. One of most general requirements in this sense, is to make sure that the system’s behaviors do not show any impulsive phenomena for the signals involved. Impulsive phenomena for implicit systems have been extensively addressed in [61, 62], while concatenability properties of behavioral hybrid systems can be found in [128].

Being in a pure denotational framework we don’t care about what the switching cause is, be it due to a given control strategy, external disturbance or an *invariance* condition being violated. Hence we analyze only the feasibility of interconnections, in terms of physical feasibility of system’s behaviors, in an effort to avoid impulsive phenomena.

Roughly speaking, interconnections of hybrid systems can be viewed as a concatenation of past and future trajectories, where the words “past” and “future” refer to the time respectively before and after the switching instant.

Below we give a formal definition of concatenations of single trajectories and whole behaviors.

**Concatenability of trajectories.** Two smooth trajectories \( w_p \in \mathcal{B} \) and \( w_f \in \mathcal{B} \) are said to be *concatenable* if \( w_p \land_0 w_f \in \mathcal{B} \).  

\(^4\)Recall the definition of the concatenation operator in the previous subsection.
15.2 Hybrid Systems and the Behavioral Approach

**Concatenability of behaviors.** The behavior $B_p$ is said to be *concatenable* with $B_f$ if, for any past trajectory $w_p \in B_p$, there exists a future trajectory $w_f \in B_f$ such that $w_p \wedge_w w_f \in B$, where $B$ is the behavior resulting from the interconnection of behaviors $B_p$ and $B_f$ according to the scheme described in Section 15.1.2.

Conditions for checking the regularity of interconnections, i.e. the avoidance of impulsive phenomena are given in [128]. In the same paper it is also shown that if the “future” interconnection (represented by behaviors in $B_f$) models a regular feedback controller \(^5\), then the past interconnection behavior $B_p$ can be concatenated with $B_f$ for any choice of the constraints induced by the past behavior. This result proves extremely important when analyzing the properties of feedback controllers for hybrid system and formally justifies their use in a hybrid interconnection scheme.

### 15.2.3 Robust Hybrid Control Using the Behavioral Approach

As we have already seen in the previous sections, hybrid systems can be conveniently modeled using a behavioral approach. Moreover, some kind of analysis of the hybrid systems properties can also be carried out in this setting, especially those related to the interconnection of hybrid modules and feasibility of trajectory concatenations.

On the other hand, the synthesis of hybrid controllers using the behavioral tools is still at its infancy. A first attempt in this direction can be found in [107], where the problem of finding a robust DES controller for a continuous plant is studied. Here the hybrid system is composed of a discrete supervisor, modeled as an automaton and a continuous plant. The control scheme is as follows: 1) the output of the plant goes through a sensor which at regular instants of time produces a symbol of the input alphabet for the discrete supervisor; 2) the automaton takes the input symbol and using its transition relation (to be determined) outputs another symbol taken from

\(^5\)See [128] for a formal definition in the behavioral framework
the output alphabet; 3) the output symbol goes through an actuator which converts it to a piecewise constant (in time) input signal for the plant to be controlled.

Due to this double sampling & quantization process, we can consider the plant behavior to take discrete (possibly finite) values at a discrete sequence of times. Indeed, if $\tilde{u}(t)$ and $\tilde{y}(t)$ for $t \in \mathbb{R}$ are the (vector) input and output signals of the plant, then the discrete equivalent plant behavior is given by sequence of symbols $u(k)$ and $y(k)$ which are respectively the output and input of the discrete event supervisor. More formally, if $W_{in}$ and $W_{out}$ are the set of input and output events, respectively, then the plant behavior is given by:

$$B_p \subseteq W_{N_0}$$

where, $W = W_{in} \times W_{out}$ and $N_0$ is the set of non negative integers.

The task of the supervisory controller is thus to restrict the plant behavior $B_p$ in such a way that the closed loop behavior is guaranteed to evolve only on acceptable signals. Formally, this is equivalent to specify a desired behavior $B_{spec}$, called the specification, and, if $B_{cl} \subseteq B_p$ is the closed loop behavior of the plant, to require that:

$$B_{cl} \subseteq B_{spec} \quad (15.1)$$

Clearly, if $B_{sup}$ is the supervisor behavior, then $B_{cl} = B_{sup} \cap B_p$.

The control problem to be addressed in this framework is to find $B_{sup}$ such that the condition (15.1) is satisfied. We observe that such a problem is univocally determined by the plant behavior $B_p$ and by the specification $B_{spec}$, thus we refer to the pair $(B_p, B_{spec})$ as a supervisory control problem. Moreover, a behavior $B_{sup}$ is called a solution to the supervisory control problem if

$$B_{cl} = B_{sup} \cap B_p \neq \emptyset$$

and $B_{cl}$ satisfies the condition (15.1).
15.2 Hybrid Systems and the Behavioral Approach

The control synthesis methodology for solving this problem is based on an abstraction process [107]: it takes the plant behavior $B_p$ and builds an abstracted behavior $B_{cal}$ such that $B_p \subseteq B_{cal}$ (i.e. the abstraction includes all possible plant behaviors plus possibly some more). The abstraction process $B_{cal}$ is defined in order to make the synthesis problem easier, i.e. the supervisory control problem $(B_{cal}, B_{spec})$ should be easier to solve than the starting problem $(B_p, B_{spec})$.

It can then be shown that the supervisory control $B_{sup}$ that solves the abstracted problem $(B_{cal}, B_{spec})$ is also a solution of problem $(B_p, B_{spec})$.

An interesting extension of the supervisory control problem just defined is to allow the controller to be robust with respect to uncertain plant parameters.

Let $\{B_\theta\}_{\theta}$ be a collection of plants $B_\theta \subseteq W^{R_0}$, that for any given $\theta \in \Theta$ are defined in terms of an automaton (possibly, but not necessarily finite). The meaning of the parameter (possibly vector valued) $\theta$ is that it models some uncertain features of the plant behavior.

The **robust supervisory control problem** is thus defined by the pair $\{\{B_\theta\}_{\theta}, B_{spec}\}$, and its solution is the behavior $B_{sup}$ such that:

$$B_{cl}(\theta) \subseteq B_{spec}$$  \hspace{1cm} (15.2)

where $B_{cl}(\theta) = B_{sup} \cap B_\theta$ for any $\theta \in \Theta$.

It can be shown that, if $B_{sup}$ is a solution of the supervisory control problem $(\cup_{\theta \in \Theta} B_\theta, B_{spec})$ then it is also a solution of the robust supervisory control problem and satisfies the condition (15.2).

Therefore, in this case also, the control problem can be addressed using the abstraction techniques used for the non-robust case, considering as the plant behavior the union of all behaviors indexed by the uncertain parameter $\theta$. 

15.3 Discussion: Behavioral Systems and Trace Algebras

Behavioral systems do have interesting affinities with trace algebras, introduced in the previous chapter. Indeed, they are both based on a denotational perspective for modeling systems as collection of traces (or trajectories).

In this section we want to show how it is possible to submerge behavioral systems in the trace algebras formalism, modeling a generic behavioral systems as a particular kind of trace structure. Recall that a behavioral system is a tuple:

\[ \Sigma = (T, W, B) \]

where:

- \( T \subset \mathbb{R} \) is the time basis (for the sake of simplicity, from now on we will consider either \( T = \mathbb{R} \) or \( T = [a, b] \) for some \( a, b \in \mathbb{R} \))
- \( W \) is the signal space, that is the set of values for the behavioral system’s variables
- \( B \subset W^T \) is the set of behaviors for the system, i.e. temporal executions of the system or traces.

We also briefly recall the main definitions of the trace algebra formalism. A trace is a tuple:

\[ x = \begin{cases} (\gamma, \delta, f) & \text{partial} \\ (\gamma, f) & \text{complete} \end{cases} \]

where: \( f : A \to \mathbb{A}^\mathbb{R} \) and, if \( v \in A \), then \( f(v) : [0, \delta] \to \mathbb{R} \) (partial) or \( f(v) : \mathbb{R} \to \mathbb{R} \) (complete).

A trace structure is a pair:
15.3 Discussion: Behavioral Systems and Trace Algebras

\((\gamma, P)\)

where:

- \(\gamma\) is a signature (set of signals)
- \(P\) is a subset of traces for that signature and it represents the set of possible behaviors of a system.

We want to show that behavioral systems ARE trace structures defined in a trace structure algebra. In order to prove this, we show that there is an isomorphism between trace structure algebra and the set of behavioral systems (conveniently augmented with suitable operators). As a first step we are going to show that behaviors are traces according to the trace algebra formalism discussed in Chapter 14.

Let \(y = (\gamma, \delta, f)\) be a trace constructed from a given BS \((T, W, B)\) in the following way:

a) \(\delta \in T\)

b) \(A = W\)

c) \(A\) and \(\gamma\) are implicitly defined by \(A\). Indeed, if -for example- \(A = \mathbb{R}^q\), then: \(A = \{x_1, \ldots, x_q\}\) and \(\gamma = (x_1, \ldots, x_q)\).

d) \(f: A \rightarrow A^\mathbb{R}\) is such that: if \(v \in A\), then: \(f(v) \in B\)

Then, with these positions, the trace structure \(Y = (\gamma, P)\) with: \(P = \{y\}\) corresponds to the behavioral system \((T, W, B)\) in a one-to-one correspondence.

Let \(g: g(Y) = (T, W, B)\) be the application that defines such a correspondence, we will show that \(g\) is an isomorphism between the trace structure algebra and a suitable “behavioral systems” algebra. However, in order to
Behavioral Approach

proceed along this path, we first need to define what we mean by behavioral system algebra (BS algebra, for short).

**BS Algebra.** A BS algebra is the set of all behavioral systems closed with respect to operators:

- projection
- renaming
- serial composition
- parallel composition

where, proj and ren are defined as in trace structure algebras on single traces, while parallel composition is semantically equivalent to the interconnection operator for behavioral systems. The serial composition is defined in terms of concatenability of trajectories induced by the operator \(\land_t\) (see Section 15.2.1).

We observe that:

- the serial composition defined in this way is different from concatenability of behaviors, which is a mixed parallel-serial operator.
- the serial composition is also semantically equivalent to serial composition in trace structure algebras.
- The closure of the operator can be easily proven.

We have thus shown that the set of behavioral systems extended with the operators of projection, renaming, serial and parallel composition is an algebra isomorphic to the trace structure algebra for hybrid systems.
Part V

Potential Interchange Formats
Chapter 16

Hybrid System Interchange Format

This Chapter describes the hybrid systems interchange format (HSIF) developed at Vanderbilt University together with the University of Pennsylvania as part of the DARPA MoBIES project. We thank Rajeev Alur, Gabor Karsai, and colleagues for making available to us the latest version of the documents on the semantics and the syntax of the proposed format [109, 71].

16.1 Introduction

HSIF models represent a system as a collection of hybrid automata called network. Each hybrid automaton is a finite state machine in which states include constraints on continuous behaviors and transitions describe discrete steps. Automata in a network communicate by means of variables. We distinguish two kinds of variables: signals and shared variables. Signals are used to model predictable execution with synchronous communication between automata. Shared variables are used for asynchronous communication between loosely coupled automata.
16.2 Syntax

16.2.1 Hybrid automaton

A hybrid automaton $H$ is a tuple $(S, s_0, V, P, T)$, where $S$ is a set of discrete locations (or states), $s_0 \in S$ is the initial location, $V$ is a set of typed variables, $P$ is a set of parameters, $T$ is a set of transitions. All sets are assumed to be finite. Note that locations are called states in other HSIF documents. To avoid confusion, the term location is used for syntactic description and the term state for semantics definition.

Variables and parameters. The set of variables of the automaton is partitioned into local variables $V_l$ and global variables $V_g$. The global variables are partitioned into signals and shared variables $V_s$. Furthermore, signals are partitioned into input $V_i$ and output $V_o$ signals. Note that an input signal of an automaton in a network can be output by another automata or it maybe global input. When considering the automaton in isolation this distinction does not matter. Local variables and output signals updated by differential equations (see below) may have an associated initial region $[imin_v, imax_v]$. When the initial region is not given a default initial value of 0 is assumed.

Parameters of a hybrid automaton are also partitioned into local and global parameters ($P_l$ and $P_g$). Parameters are assigned fixed values when an automaton is created, and the values remain constant throughout all executions.

Predicates and functions. Behaviors of an automaton are given in terms of predicate $g(x_1, \ldots, x_n)$ and function $f(x_1, \ldots, x_n)$, where $x_i$ is a variable or parameter of the automaton. A function or a predicate may be specified by a mathematical expression, a procedure to compute the function, or in a tabular form.
Locations. A location contains a set of differential and algebraic equations and one invariant that define continuous behaviors while the automaton is executing in the location.

- A differential equation has the form $\dot{x} = f(x_1, \ldots, x_n)$. $\dot{x}$ is understood as the first derivative of $x$ with respect to time.
- An algebraic equation has the form $x = f(x_1, \ldots, x_n)$.
- The invariant is a predicate $i(x_1, \ldots, x_n)$.

For differential and algebraic equations, $x$ on the left-hand side can be either a local variable or an output signal.

The left-hand side variables of the equations in a location are pairwise distinct. Local variables and output signals that do not have an equation in some location are assumed to be constant while the automaton is in that location. That is, the equation $\dot{x} = 0$ is assumed for every such variable. Since shared variables may be updated only by the transitions of the automaton, equation $\dot{x} = 0$ is also assumed for each shared variable $x$.

For each location, the set of algebraic equations must have a uniquely well-defined solution. This is guaranteed by requiring that the variable dependency relation for algebraic equations is not circular. This can be checked by constructing a graph of dependencies between the variables that are updated by algebraic equations with edges of the form $x_i \to x$, for $i = 1 \ldots n$, whenever there is an algebraic equation $x = f(x_1, \ldots, x_n)$, and then ensuring that the dependency graph is acyclic.

Transitions. A transition of a hybrid automaton is a tuple $(s, g, \alpha, s')$, where $s$ and $s'$ are the source and target locations of the transition. The guard $g$ is a predicate. The guard has to evaluate to true before the transition can happen. The action $\alpha$ is executed when the transition happens. The action is a sequence (possibly empty) of assignments to the automaton
16.2 Syntax

variables, executed atomically when the transition occurs. An assignment has the form \( x = f(x_1, \ldots, x_n) \), where \( x \) cannot be an input variable.

To provide for the synchronous execution of discrete steps, we extend each location of an automaton with a default transition. The default transition has an empty action and its guard is equal to the invariant of the location. The reason to have this transition is that the automaton must participate in a discrete step of the network, but it does not have to execute any of its regular transitions until its invariant is violated. As long as the invariant of the active location of the automaton is satisfied, the automaton has a choice of executing an enabled regular transition or the default transition. When the invariant is violated, the default transition is not enabled, and the automaton has to execute a regular transition, if there is an enabled one.

Summary of variable uses. Figure 16.1 summarizes the kinds of variables in an automaton and their uses.

16.2.2 Network of hybrid automata

A network is a tuple \( \langle HA, V, P, C \rangle \), where \( HA \) is a set of hybrid automata, \( V \) is a set of variables, \( P \) is a set of parameters, and \( C \) is an input constraint. The set \( V \) is partitioned into signals, local variables \( V_l \), and shared variables \( V_s \). Signals are further partitioned into input signals \( V_i \) and output signals \( V_o \).

The following consistency requirements are assumed:

- For each automaton \( h \in HA \), \( V_s^h = V_s \) and \( P_g^h = P \). That is, the shared variables and global parameters are available to all automata in the network.

- Each output signal is updated by some automaton, that is, \( \bigcup_{h \in HA} V_o^h = V_o \), and signals output by different automata are pairwise disjoint.
• Input signals of the network may be used only as inputs by the automata, however, some inputs of an automaton may be signals output by other automata. That is, $V_i \subseteq \bigcup_{h \in HA} V_h^i$.

• Local variables of the network are the union of local variables of the automata, $\bigcup_{h \in HA} V_h^i = V_i$, and local variables of different automata are pairwise disjoint.

The constraint $C$ is a predicate over the input variables and parameters of the network and is used to constrain the input variables of the network.

**Automata dependencies.** A graph of dependencies between automata in the network can be built as follows: there is an edge $h_1 \rightarrow h_2$ if there is a signal $x$ that is an output signal of $h_1$ and an input signal of $h_2$ used in an algebraic equation, invariant, transition guard, or assignment in a transition action. To ensure predictable executions of the network the graph
of dependencies must be acyclic.

16.3 Semantics

Instantiated vs. parameterized automata. This section discusses the semantics of instantiated automata and networks of automata. Given a network, an instantiated network is obtained by assigning a fixed value to each parameter in every automaton in the network, and replacing each occurrence of the parameter with the chosen value.

16.3.1 Semantics of a single automaton

Flows and actions. Given a set of variables $V$, we use $Q_V$ to denote a valuation of the variables in $V$, i.e., a function from $V$ to the set of values for the variables in $V$. We omit $V$ when it is understood from the context. The value of a variable $x$ in the valuation $Q$ is denoted as $Q(x)$.

A continuous step of an automaton or a network is defined by a flow $f$ over the set of the automaton variables $V$, which is a differentiable function from a time interval $[0, T)$ ($T > 0$) to $2^{Q_V}$, the set of valuations of the automaton variables. By $f_x(t)$ we will denote the value of variable $x$ in the valuation $f(t)$. Given a flow $f$ over $V$ and a set of variables $V' \subseteq V$, a restriction of $f$ to $V'$, denoted $f \downarrow V'$, is the flow over $V'$ with the same domain as $f$, such that $(f \downarrow V')_x = f_x$ for all $x \in V'$.

A discrete step of an automaton or a network is defined by an action over a set of variables $V$, which is a function $\alpha : 2^{Q_V} \rightarrow 2^{Q_V}$. A restriction of an action to a subset of variables is defined similarly to the restriction on flows.

State of a hybrid automaton. A state of the hybrid automaton $H = \langle S, s_0, V, P, T \rangle$ is a tuple $\langle s, Q_V \rangle$, where $s \in S$ is a location and $Q_V$ is a valuation of the variables $V$. An initial state of the automaton is $\langle s_0, Q_0 \rangle$, where the initial valuation $Q_0$ is such that
• the value of each local or output variable lies within its initial region;

• the invariant of $s_0$ is satisfied; and

• all algebraic equations in $s_0$ are satisfied.

**Executions of an automaton.** An execution of an automaton is a finite or infinite sequence of steps starting in an initial state. Each step in an execution is either a continuous step of the automaton, a discrete step of the automaton, or an environment step. Continuous steps do not affect the current location of the automaton. During a continuous step, time advances at the same rate in the automaton and its environment. Because of this, all of the automaton variables are updated simultaneously during a continuous step. Local and manipulated variables are updated according to the dynamics specified in the current location of the automaton. Input variables are updated by the environment of the automaton. An environment step is a discrete step by another automaton in the network or by the environment of the network. An environment step can change the values of global variables of the automaton.

Discrete and continuous steps in an execution alternate, and between each discrete or continuous step there is an environment step, as illustrated below:

$$(s_1, Q_1) \xrightarrow{f_1} (s_1, Q'_1) \xrightarrow{e_1} (s_1, Q''_1) \xrightarrow{\alpha_1} (s_2, Q_2) \xrightarrow{e_2} (s_2, Q'_2) \xrightarrow{f_2} \ldots$$

**Continuous steps of an automaton.** The automaton has a continuous step $(s, Q) \xrightarrow{f} (s, Q')$ if $f(0) = Q$, $\lim_{t \to T} f(t) = Q'$, and for each time instance $t \in [0, T)$ the following conditions hold:

• If $s$ has a differential equation $\dot{x} = h(x_1, \ldots, x_n)$, then the first derivative with respect to time of $f_x(t)$ equals to $h(f_{x_1}(t), \ldots, f_{x_n}(t))$.

• If $s$ has an algebraic equation $x = h(x_1, \ldots, x_n)$, then the value of $f_x(t)$ equals to $h(f_{x_1}(t), \ldots, f_{x_n}(t))$. 
16.3 Semantics

- The invariant of \( s \) is satisfied; i.e., \( i(f_{x_1}(t), \ldots, f_{x_n}(t)) \) evaluates to true.

**Discrete steps of an automaton.** A discrete step of an automaton is a result of taking a transition of the automaton from one location to another. The automaton has a discrete step \( \langle s, Q \rangle \xrightarrow{\alpha} \langle s', Q' \rangle \) if the automaton contains the transition \( \langle s, g, \alpha, s' \rangle \) such that the following conditions hold:

- the guard of the transition is satisfied by \( Q \), that is, \( g(Q(x_1), \ldots, Q(x_n)) \) is true;
- the action of the transition transforms the source valuation into the target valuation, \( \alpha(Q) = Q' \); and
- the invariant of \( s' \) is satisfied by \( Q' \).

**Environment steps.** Environment steps in the execution of an automaton are discrete steps that occur in the environment of the automaton. There is an environment step \( \langle s, Q \rangle \xrightarrow{e} \langle s, Q' \rangle \) whenever \( Q(v) = Q'(v) \) for each local variable \( v \). Note that the values of local variables and the current location of the automaton is unchanged.

16.3.2 Semantics of the network

**State of the network.** A state of the network of automata \( h_1, \ldots, h_n \) is a tuple \( \langle \langle s_1, \ldots, s_n \rangle, Q_V \rangle \), where \( \langle s_1, \ldots, s_n \rangle \in S_{h_1} \times \ldots \times S_{h_n} \) are locations of the automata of the network.

An initial state of the network is such that, for each \( i \), \( \langle s_i, Q_V \downarrow V_{h_i} \rangle \) is an initial state of the automaton \( h_i \) and the constraint \( C \) is satisfied.

**Continuous steps of the network.** The network has a continuous step \( \langle \langle s_1, \ldots, s_n \rangle, Q \rangle \xrightarrow{f} \langle \langle s_1, \ldots, s_n \rangle, Q' \rangle \) if the following conditions hold:

- for all \( t \in [0, T] \), the constraint \( C \) is satisfied by \( f(t) \).
Hybrid System Interchange Format

- for each hybrid automaton \( h_i \), there is a continuous step \( \langle s_i, Q \downarrow V^{h_i} \rangle \xrightarrow{f_i|V^{h_i}} \langle s_i, Q' \downarrow V^{h_i} \rangle \).

Note that continuous steps are taken synchronously by all automata and the time progresses at the same rate in all automata.

**Discrete steps of the network.** The network has a discrete step

\[ \langle \langle s_1, \ldots, s_n \rangle, Q \rangle \xrightarrow{\alpha} \langle \langle s'_1, \ldots, s'_n \rangle, Q' \rangle \]

if there exists an ordering of the automata in the network consistent with the partial order induced by the automata dependency graph, say \( h_1, \ldots, h_n \) without loss of generality, and there exists a sequence of steps

\[ \langle \langle s_1, s_2 \ldots, s_n \rangle, Q_0 \rangle \xrightarrow{\alpha_1} \langle \langle s'_1, s_2 \ldots, s_n \rangle, Q_1 \rangle \xrightarrow{\alpha_2} \langle \langle s'_1, s'_2 \ldots, s_n \rangle, Q_2 \rangle \xrightarrow{\alpha_3} \ldots \xrightarrow{\alpha_n} \langle \langle s'_1, \ldots, s'_n \rangle, Q_n \rangle \]

such that

- \( Q = Q_0 \) and \( Q' = Q_n \);
- The action can be decomposed into \( n \) simpler actions \( \alpha = \alpha_1 \circ \alpha_2 \circ \ldots \alpha_n \); and
- For each automaton \( h_i \), there exists a discrete step \( \langle s_i, Q_{i-1} \downarrow V^{h_i} \rangle \xrightarrow{\alpha_i} \langle s'_i, Q_i \downarrow V^{h_i} \rangle \), where \( Q_{i-1} \) and \( Q_i \) agree on all variables not in \( V^{h_i} \).

This definition means that a discrete step of the network is the result of a sequence of discrete steps of each of the automata taken in an order that respects automata dependencies.

**Executions of the network.** An execution of the network is a finite or infinite sequence of steps \( \langle \langle s_1, \ldots, s_n \rangle, Q \rangle \rightarrow \langle \langle s'_1, \ldots, s'_n \rangle, Q' \rangle \rightarrow \langle \langle s''_1, \ldots, s''_n \rangle, Q'' \rangle \rightarrow \ldots \), starting in an initial state. Each step of the network consists of a continuous step followed by a discrete step. Precisely, the step \( \langle \langle s_1, \ldots, s_n \rangle, Q \rangle \rightarrow \langle \langle s'_1, \ldots, s'_n \rangle, Q' \rangle \) in an execution of the network is possible if there exist a state \( \langle \langle s_1, \ldots, s_n \rangle, Q_i \rangle \) such that the network has a continuous step.
16.3 Semantics

\( \langle \langle s_1, \ldots, s_n \rangle, Q \rangle \xrightarrow{f} \langle \langle s_1, \ldots, s_n \rangle, Q' \rangle \) for some flow \( f \) and a discrete step \( \langle \langle s_1, \ldots, s_n \rangle, Q' \rangle \xrightarrow{\alpha} \langle \langle s'_1, \ldots, s'_n \rangle, Q'' \rangle \) for some action \( \alpha \).

An exception is the last step of a finite execution, which consists of only a discrete step. In this case, the network reaches a state \( \langle \langle s_1, \ldots, s_n \rangle, Q \rangle \), in which there is an automaton \( h_i \) such that there are no enabled transitions in the state \( \langle s_i, Q \downarrow V^{h_i} \rangle \) and the invariant of \( s_i \) is violated. In this state, the continuous step cannot be extended and a discrete step is impossible.

**Computation of executions.** The initial state for an execution can be obtained in the following way:

1. assign values to the network inputs to satisfy the input constraint of the network;
2. assign values to the shared variables and output and local variables updated by differential equations in \( s_0 \) according to the initial regions of each variable;
3. assign values to variables updated by algebraic equations in \( s_0 \) computing the right-hand sides of each equation in the order of dependencies between automata and in the order of algebraic dependencies within \( s_0 \).

The states of the network during a continuous step can be numerically computed by selecting an ordering of automata consistent with the partial order induced by the dependency graph of automata, and then, considering automata in the selected order, first solve the differential equation in the active state of the automaton, then solve the algebraic equations (in the order consistent with the graph of variable dependencies in the automaton).

A discrete step of the network can be computed by computing discrete steps of each automaton independently, in the order respecting automata dependencies, and concatenating the steps.
Any ordering of automata allowed by the dependency order can be used for computing continuous steps. Any such ordering will yield the same result. Therefore, an ordering can be chosen prior to the execution and used in all steps.

However, if shared variables are used, then different orderings of automata may produce different results when computing discrete steps. This is because dependencies between shared variables are not captured by the graph of dependencies between automata. Therefore, an ordering (that is consistent with the dependency order) may be chosen randomly before each discrete step during simulation. For exhaustive state-space exploration, all orderings (that are consistent with the dependency order) need to be considered.

16.4 Communication in the Network

Pictorial representation of network. In the examples below, a network is represented as a collection of rectangular boxes with ports. Ports represent input and output variables. Arrows connecting ports represent data connections between automata. Connections that constitute dependencies between automata are shown as solid lines, while the rest are shown as dashed lines. A location in an automaton is represented as a rounded box with its equations and invariants shown inside the rounded box. Invariants are shown in braces to distinguish from equations. Transition labels are shown as $g \rightarrow \{x_1 = e_1; x_2 = e_2\}$.

Signals and shared variables. During execution of the network, every signal or local variable $v$ is represented as a function (not necessarily continuous) from time to the value of $v$. In this way, all automata that input $v$ will observe the same value of $v$, since all steps proceed in the order of dependencies. For shared variables, this may not be true since a shared variable may be assigned multiple times during a discrete step of the network.
16.4 Communication in the Network

![Diagram of automata and shared variables](image)

Figure 16.2: Shared variables and Nondeterminism.

However, during each continuous step, all automata will observe the same value for a shared variable.

Dependencies between signals are captured to impose an execution order between automata. Each signal must either be a global input to the network or be modified by exactly one automaton. Communication by means of signals between automata in a well-formed network does not introduce any nondeterminism in the execution of the network. By contrast, dependencies between shared variables do not constrain executions of the network and multiple automata can modify shared variables. Therefore, communication using shared variables may be nondeterministic if two automata modify a shared variable in the same discrete step.

The network of Fig 16.2 illustrates nondeterminism introduced by the use of shared variables. The automaton $h_1$ provides the signal $x$ to $h_2$. $h_1$ updates the shared variable $y$ and $h_2$ updates the shared variable $z$. The automaton $h_3$, which reads $y$ and $z$, does not depend on either $h_1$ or $h_2$. Therefore, in a discrete step, $h_3$ can proceed (i) before $h_1$, in which case it will have to use the default transition, (ii) after $h_1$ but before $h_2$, in which case only one of its transitions will be enabled, or (iii) after both $h_1$ and $h_2$ and have both of its transitions enabled.
Inputs used in differential equations are not dependencies. The arguments of differential equations are not considered when constructing dependencies between automata. If an input is used only in a differential equation, say $\dot{y} = f(x)$, then $y(t)$ does not depend on $x(t)$ and thus does not constitute a dependency. Figure 16.3 reports the example of a well-formed network of two single-state automata.

There is a dependency from $h_1$ to $h_2$, but the dependency graph is acyclic and the state at time $t$ can be computed by solving the differential equation in $h_1$ first and then updating $x$ using the algebraic equation. It is easy to see that the differential equation can be rewritten to eliminate the dependency on $x$: $\dot{y} = f_1(f_2(y))$.

Events. A useful synchronization mechanism is based on events. Events are signals that are instantaneously broadcast by an automaton in the network to all other automata. Events do not carry values but may be used to trigger discrete transitions in other automata. In the proposed approach, events are not supported explicitly. Instead, events are modeled as boolean signals that have the value false during all continuous steps (specified as an algebraic equation in all locations of an automaton) and can be set to true by a discrete transition. Then, automata that depend on this signal can use the signal to trigger their own discrete transitions. The next continuous step, however, will reset the signal to false. An example is illustrated in Figure 16.4.
16.4 Communication in the Network

![Diagram of Events and Continuous Transitions]

Figure 16.4: Events and continuous transitions.

**Input constraints.** Consider the network of two single-state automata that represent two tanks that are being filled via the same pipe. The pipe is not captured by the model explicitly. The flows into the two tanks are the inputs to the network, but the sum of flows is bounded by the capacity of the pipe. Using inputs $f_1$ and $f_2$ to represent flows into each tank and $l_1$ and $l_2$ as the output signals of the two automata, we have differential equations in the two automata $\dot{l}_1 = f_1$ and $\dot{l}_2 = f_2$, while the input constraint is $f_1 + f_2 \leq p$ (for some parameter $p$).

**Purely synchronous network.** In a purely synchronous network (i.e., the network without any shared variables), automata communicate only by means of signals. The requirement that dependencies between automata be acyclic imposes restrictions on communication between automata. Consider the two components that communicate by means of events (i.e., discrete signals as described above). One automaton sends a message, which is represented by the event $msg$, and expects to receive an acknowledgement as event $ack$. One HSIF model for such a system is the network of Figure 16.5.

Notice that events $msg$ and $ack$ are always temporally separated. That is, at any time instance, only one of the two events can be updated. Still, the dependency graph, which is determined statically, has a cycle and this network is rejected by the current semantics. There are two alternatives to the chosen semantic approach:

- A dynamic notion of dependencies, where the dependency graph is required to be acyclic in any reachable state. However, ensuring that the
system satisfies the dynamic requirement is an undecidable problem, in general.

- A fixed-point semantics can be given to signals as it is done in Stat-echarts [76] and some other synchronous languages. However, this makes the approach quite more complex.

**Purely asynchronous network.** Arguably, the system like this is more naturally modeled by asynchronous communication. If signals $msg$ and $ack$ are replaced by shared variables, the resulting system is purely asynchronous. This particular system does not introduce any nondeterminism because updates to the shared variables are temporally separated. In general, it is hard to predict whether or not nondeterminism has been introduced due to shared-variable communication.

**Asynchronous network of synchronous networks.** It is possible to combine the two modes of communication in the same network. Consider
16.5 Discussion

Figure 16.6: A clustered network.

a case where the graph of dependencies between automata consists of several disconnected subgraphs, called clusters. Within a cluster, automata communicate via signals, whereas communication between automata in different clusters is only by shared variables. Figure 16.6 sketches the structure of such a clustered network. Clustered networks are suitable for modeling communicating autonomous systems such as robot formations. For example, each robot may be modeled as a synchronously communicating cluster and uses shared variables to communicate with other robots.

16.5 Discussion

Before evaluating HSIF, we need to clarify the differences between modeling languages and interchange formats. The goal of a modeling language is to enable the formal representation of selected aspects of a system. As such, a modeling language is always restrictive (only selected aspects are modeled), formal (has well defined concrete syntax, abstract syntax, and semantics) and unambiguous. The goal of an interchange format is to communicate models among tools using different modeling languages. Accordingly, interchange formats are not restrictive (all syntactically and semantically sound models can be interchanged), syntax free (allow tools to use different domain specific concrete syntax) and unambiguous. There are two opposite
Hybrid System Interchange Format

approaches for defining model interchange formats. In the semantic free approach the interchange format is nothing more but a common transfer format for models. In this case model transformers (semantic translators) must provide pair wise mapping among the tool models based on their shared portion of the semantics. In the semantically inclusive approach, a common modeling language and transfer format for is defined for model interchange, which has broad enough semantics to allow exporting and importing individual tool models to and from this shared language. HSIF, in its current form, clearly moved toward the second approach. In its current stage, the HSIF specification has the following unresolved issues:

1. It is semantically too rich to become a semantic free common transfer format, but semantically too restrictive to become a common modeling language. For example, it prevents “by-construction” zero-time loops among FSM-s to eliminate the risk of non-deterministic behavior stemming out of a combination of deterministic subsystems.

2. It is syntactically too restrictive by lacking support for hierarchical FSMs. This can be problematic as other models often allow the creation of an hierarchical network of FSM. For instance, exporting a HYVISUAL hierarchical model into HSIF requires that the hierarchy of each FSM is flattened first, a transformation that is not easy to reverse.

It must be noted that HSIF is not a completed proposal, but rather a work in progress. It helped MoBIES researchers to understand some of the fundamental problems in forming a standardized semantics for tools and to understand at least some of the hard issues of different kinds of semantics in modeling languages. The jury is still out to determine whether interchange formats will evolve toward semantic free or semantically inclusive direction. We argue that the elimination of semantically unsound behaviors should be up to the tools, particularly the synthesis tools, and not to the interchange
16.5 Discussion

format. Otherwise there is a definite risk for the format of not being able to accept the description of legitimate systems in tools where a larger set of behaviors is accepted. While we advocate that tools should be very careful in adopting liberal models, we believe that the design methodology should be enforced by tools not by interchange formats.
Chapter 17

Metropolis

METROPOLIS is an ambitious project supported by the GSRC (Gigascale System Research Center), the CHESS (Center for Hybrid and Embedded Software Systems) and grants from industry. It aims to support design from specification to implementation on hardware and software platforms. One of its most important features is the mechanism to represent and manipulate designs. This mechanism is based on the definition of a particular model of computation called METROPOLIS meta-model that sits above the commonly used models of computation such as FSMs and Data Flow graphs, in the sense that these models can be obtained by refining the meta-model representation of a design.

The meta-model is also intended to provide support for “inventing” novel models of computation wherever they may be needed. We have already seen instances where the introduction of models of computation that are specific to the design problem at hand can improve the design process in a substantial way. The meta-model is general enough that can be used to represent not only behaviors like many of the approaches presented in the reports that are part of the DHS work-package of Columbus, but also implementation architectures thus allowing a designer to live his/her activity in only one framework.
17.1 The Metropolis Methodology

In the following sections we present the key concepts of the Metropolis methodology and the Metropolis meta-model. For a more complete discussion of Metropolis see [17, 16]. The complete definition of the Metropolis meta-model is given in [129], Finally [97] discusses the modeling of architectural resources in Metropolis.

17.1 The Metropolis Methodology

To support complex system designs, the design methodology and associated tools and flows must start capturing the design specifications at the highest level of abstraction and proceed towards an efficient implementation. Critical decisions must be made about the architecture of the system and the choice of components to implement the architecture and carry out the computation and communication tasks associated with the overall structure of the design.

Metropolis proposes a design methodology for embedded system design, i.e. embedded software design carried out paying close attention to the selection and configuration of the platform (both hardware and software) on which the application software runs. The methodology is based on the following key aspects. First of all, it leaves the designer relatively free to use the specification mechanism (graphical or textual language) of choice, as long as it has a sound semantic foundation (model of computation [54]). Secondly, it uses a single formalism to represent both the embedded system and some abstract relevant characteristics of its environment and implementation platform [120]. Finally, it separates orthogonal aspects [93], such as:

- **Computation and communication.** This separation is important because:
  
  - refinement of computation is generally done by hand, or by compilation, or by scheduling, and other complex techniques;
refinement of communication is generally done by use of patterns (such as circular buffers for FIFOs, polling or interrupt for hardware to software data transfers, and so on).

- **Functionality and architecture**, or functional specification and implementation platform, because they are often defined independently, by different groups (e.g., video encoding and decoding experts versus hardware/software designers in multimedia applications). Functionality (both computation and communication) is mapped to architecture in order to specify a given refinement for automated or manual implementation.

- **Behavior and performance indices**, such as latency, throughput, power, energy, and so on. These are kept separate because:

  - when performance indices represent constraints they are often specified independently from the functionality by different engineering teams (e.g., control engineers versus system architects for automotive engine control applications);

  - when performance indices represent the result of implementation choices, they derive from a specific architectural mapping of the behavior.

All these separations result in better re-use, because they decouple independent aspects, that would otherwise tie, e.g., a given functional specification to low-level implementation details, or to a specific communication paradigm, or to a scheduling algorithm. It is very important to define only as many aspects as needed at every level of abstraction, in the interest of flexibility and rapid design space exploration. They also allow extensive use of synthesis, system-level simulation and formal verification techniques in order to speed up the design cycle.
17.2 The Metropolis Meta-Model

Other fundamental aspects are the abilities to (1) specify rather than implement, (2) execute reasonably detailed but still fairly abstract specifications, and (3) use the best synthesis algorithms for a given application domain and implementation architecture. For these reasons in Metropolis the concurrency available at the specification level is represented explicitly in the form of multiple communicating processes. An executable representation for the computation processes and the communication media can also be used in order to allow both simulation and formal and semi-formal verification techniques to be conveniently exploited.

As a final step towards the implementation layer, Metropolis also features a restriction of the executable representation with respect to a full-fledged programming language such as C, C++ or Java, in order to improve both analyzability and synthesizability.

17.2 The Metropolis Meta-Model

The other important principle that can be identified as a general feature is how to handle the communication mechanism among the library elements. The term component (or communication) based design refers to the primary importance given to the appropriate selection of interfaces and communication protocols. If indeed interfaces and communication mechanisms are carefully identified and specified, then design re-use is greatly simplified and enhanced.

The goal of the Metropolis project is to cope with these problems in a unified framework. The idea is to provide an infrastructure based on a model with precise semantics, yet general enough to support the models of computation [98] proposed so far and, at the same time, to allow the invention of new ones. The model, called meta-model for its characteristics, is capable not only of supporting functionality capturing and analysis, but also architecture description and mapping of functionality to Architectural elements. Since the model has a precise semantics, it can be used to support a number
of synthesis and formal analysis tools in addition to simulation. METROPOLIS does not pretend to dictate the use of a particular design language nor of a unified flow for all applications: the infrastructure is built so that it offers a translation path from specification languages to the meta-model. In addition, mechanisms are provided to allow the integration of external tools, thus alleviating the problems of building flows with tools that are developed independently and with different semantic models. Finally, the structure of the meta-model has been carefully selected to favor separation of concerns, such as function and architecture, communication and computation, and to support the rigorous successive refinement approach advocated by platform-based design. Design constraints are captured using a logic language and execution constraints. For example priorities can also be specified independently of the functional model.

As it has been briefly explained above, in METROPOLIS behaviors, architectures, and environments are all specified using the formalism called the meta-model [17]. To specify any of these there, one needs a capability of describing the following aspects: actions, constraints, and their refinements. We will see in the rest of this chapter that METROPOLIS does have such a capability, allowing to model, analyze and design systems in an extremely general framework.

17.2.1 Behaviors as actions

A behavior can be defined as concurrent occurrences of sequences of actions. Some action may follow another action, which may take place concurrently with other actions. The occurrences of these actions constitute the behavior of a system that the actions belong to.

An architecture can be defined as the capacity of actions it can provide. Some actions may realize arithmetic operations, while others may transfer data. Using these actions, one can implement the behavior of the system. A description of actions can be made in terms of computation, communi-
17.2 The Metropolis Meta-Model

cation, and coordination. The computation defines the input and output sets, a mapping from the former to the latter. The communication defines a state and methods. The state represents a snapshot of the communication. For example, the state of communication carried out by a stack may represent the number of elements in the stack and the contents of the elements. The communication methods are defined so that they can be used to transfer information. The methods may evaluate and possibly alter the communication state. In the case of a stack, methods called pop and push may be defined. Actions for computation and communication often need to be coordinated. For example, one may want to prohibit the use of the pop method, while an element is added to the stack by the push method.

In the meta-model, special types of objects called process and medium are used to describe computation and communication, respectively. For coordination, one can write formulas in linear temporal logic [111] to specify the coordination, or use schedulers to describe a particular algorithmic implementation of constraints. When an action takes place, it incurs cost. Cost for a set of actions is often subject to certain constraints. For example, the time interval between two actions may have to be smaller than some bound, or the total power required by a sequence of actions may need to be lower than some amount. The meta-model mechanism let us define a quantity such as time or power, associate it with actions, and put constraints on it in form of predicate logic.

17.2.2 Action automata

The set of actions of a netlist consists of actions that each process in the netlist can take. The set of actions that a particular process can take includes:

1. all statements that the process can execute,

2. all function calls that the process can make,
3. all assignment expressions that the process can execute, i.e. all expressions of the form \( x \sim expr \), where \( x \) is a variable, \( expr \) is an expression, and \( \sim \) is one of the allowed assignment operators.

4. all top-level expressions that the process can execute: expressions that are statements, right-hand sides of assignment expressions, and expressions that appear as arguments in function calls.

With each action \( a \) we associate two events, \( a^+ \), indicating the start of execution of \( a \), and \( a^- \) indicating the end. The cross-product of all the sets of events in the system is called the set of event vectors.

**State variables.** State variables are defined for objects of communication media and processes. For an object of a communication medium in a given netlist, the instances of the fields of the medium are called the state variables of the object. For a process object in the netlist, consider the set of functions that can be called by the object. Specifically, the function thread is in the set. Also, if a function \( f \) is in the set, all the functions that can be called from \( f \) are in the set. The instances of variables declared in the functions in this set, and instances of the fields of the process constitute the state variables of the object. In addition, there is a state variable for each action that is also an expression. Intuitively, this state variable is used to temporarily store the value of the expression. Notice that sets of process state variables are disjoint. The set of all state variables is called the memory, and an assignment of values to all state variables is called a memory state.

**Action automata.** The execution semantics of a meta-model netlist is defined by the language of action automata which are defined over the alphabet containing event vectors. There is at least one action automaton for each action of each process. Action automata are quite general instances of generic automata, however they possess an interesting particularization in that they have associated a so called care set. Intuitively an action automata
17.2 The Metropolis Meta-Model

Figure 17.1: A netlist of processes and media. The object around the process is the environment, which is modeled by a medium in this example.

controls the events in its care set, but it is not affected by any other events. For a formal definition, see [129].

17.2.3 Processes and media

Fig. 17.1 shows a network of meta-model objects, where rectangles represent processes and circles represent media. It consists of two independent data streams. In each stream, the two processes on the left send integers, and the process on the right receive them. The medium in the middle defines the semantics of this communication.

Processes are active objects in the sense that they take their own actions concurrently with those of other processes. The specification of a process of Figure 17.1 is given in Figure 17.2. The syntax is similar to that of JAVA. A process always defines at least one constructor and exactly one function called thread, the top-level function to specify the behavior of the process. We call it thread because it is given as a sequential program to define a
sequence of actions that the process takes. A process interacts with other objects through ports. A **port** is a special kind of field with the type being an interface. An interface declares a set of functions with the types of their inputs and outputs, without implementing them. A process may access its ports and call functions declared in the corresponding interfaces. The interfaces used in Fig. 17.1 is shown on the right-end side of Figure 17.2. The keyword **update** indicates that the corresponding function may change the state of a medium that implements the interface. Similarly, **eval** indicates that the function may only evaluate the state but not change it.

Figure 17.3 shows the specification of one of the media used in Figure 17.1. A medium implements interfaces by providing code for the functions of the interfaces. As with processes, a medium may define fields and functions, where some of the fields may be ports. They may not be accessed by objects other than itself. The only exception is that a function of an interface implemented by the medium object may be called by an object that has a port connected to the medium object. Such connections are specified in netlists, where a port may be connected to a single object of a medium type which implements the port’s interface. With such a connection, a call
medium IntM implements IntWriter,IntReader,IW,IR,IC,IS,IN {
  int storage, space, n;
  IntX() { space =1; n = 0;}
  update void writeInt(int data) {
    await (space > 0; this.IW, this.IS; this.IW)
    await (true; this.IC, this.IS, this.IN; this.IC) {
      space = 0; n=1;
      storage = data;
    }
  }
  update int readInt() {
    await (n > 0; this.IR, this.IN; this.IR)
    await (true; this.IC, this.IS, this.IN; this.IC) {
      space = 1; n=0;
      return storage;
    }
  }
  eval int space() {await(true; this.IW, this.IC; this.IS) return space; }
  eval int n() {await(true; this.IR, this.IC; this.IN) return n; }
}
/* Interfaces used inside IntM */
interface IW extends Port {}
interface IR extends Port {}
interface IC extends Port {}
interface IS extends Port {}
interface IN extends Port {}
of a function of the interface through the port will execute the code of the function provided in the medium.

Instances of the `await` statement often appear in functions of both processes and media. This is one of the most important constructs, since it is the only one in the meta-model to specify synchronization among processes in the execution code, in addition to the logic formulae described later. It is used in a situation where a process needs to wait until a certain condition holds, and once the condition becomes true, the process takes a sequence of actions. We call such a sequence critical section. Further, it is possible to specify actions that should not be taken by other processes while the process is in the critical section.

Consider the code of the process shown in Figure 17.4. Inside the keyword `await`, the parentheses specify a condition to be checked and actions to be excluded. This is followed by a specification of the critical section, inside the braces in the example. The parentheses consist of three sections separated by semicolons, which are called the guard, test list, and set list respectively.

The guard, a Boolean expression, specifies the condition that must hold when the execution of the critical section begins. In Figure 17.4, an interface function `n()` is called in the guard. Suppose that this function returns the number of data elements available in the storage of the corresponding medium object. Then the guard becomes true when both of the media connected to the ports of the process have at least one data element respectively. This is the semantics used in dataflow networks. In general, `await` is capable of modeling different semantics by using different guards. For example, if the conjunction used in the guard in Fig. 17.4 is replaced by disjunction, then the guard becomes true if at least one of the media has data, which is the semantics employed in discrete event systems.

The test list specifies that must not be executing when the critical section starts. The set list specifies actions that should not start while the critical
17.2 The **Metropolis Meta-Model**

```java
process Y {
    port IntReader port0;
    port IntReader port1;
    port IntWriter port2;
    ...
    void thread () {
        int z;
        while (true) {
            await ((port0.n() >0 && port1.n() >0);
            port0.IntReader, port1.IntReader;
            port0.IntReader, port1.IntReader) {
                z = foo(port0.readInt(),port1.readInt());
            }
            port2.writeInt(z);
        }
        int foo(int x, int y) {...}
    }
}
```

Figure 17.4: Process using an **await** statement.
section is executed. For example, in Figure 17.4, both test list and set list contain an element specifying IntReader interface of the medium connected to port0 indicating that the critical section is mutually exclusive to the set of actions that contains all function calls made by other processes to that medium through IntReader interface (e.g. calls readInt() of that medium). If there are more than one critical sections that can be entered, the process nondeterministically chooses exactly one of them to execute, and exits the entire await statement when the execution of the chosen section is completed.

17.2.4 Refinement

Once objects are instantiated and connected, some of them may be refined further to provide details of the behavior. Such details are often necessary when particular architecture platforms are considered for implementation. For example, the specification of Figure 17.1 assumes communication with integers, and each medium has a storage of the integer size. However, the chosen architecture may have only a storage of the byte size, and thus the original communication needs to be implemented in terms of byte-size communication. In the refinement, the semantics of the communication must remain the same as the original when observed from the processes, i.e. the processes can issue functions of reading and writing integers under the exclusion specified in the original medium.

Figure 17.5 illustrates a refinement of the medium of Figure 17.3. In general, a refinement of an object is specified as a netlist of objects, and a refinement relation between the netlist and the original object is specified using the refine statement. Such a netlist is often specified by a designer who defines architecture platform, and is stored in a library together with the original object being refined. Since this designer is in general different from a system designer who instantiates the platform objects to specify his system, it is usually unknown how the original object is used in a particular
17.2 The Metropolis Meta-Model

Figure 17.5: A refinement of the medium IntM (left). A refinement of the netlist of Figure 17.1 using the refinement netlist of the medium IntM (right). The objects shown with names are instantiated in the netlist DoubleStream, while those without names are created inside the refinement netlists on the left-end side.
system. The system designer, who first instantiates the original object to constitute his system behavior, then chooses a particular refinement netlist for the object to meet cost/performance goals.

In Figure 17.5, the refinement netlist $\text{RefIntM}$ contains three types of media. $\text{ByteM}$ is the medium with a byte storage. It implements interfaces called $\text{ByteWriter}$ and $\text{ByteReader}$, which are identical with $\text{IntWriter}$ and $\text{IntReader}$ except that the size of data is byte. One object of this type is used in the refinement, which may be provided externally. $\text{ByteW}$ implements the $\text{IntWriter}$ interface so that each of its objects is connected from a port of an object (such as $P0$ in Fig. 17.1) that originally accesses $\text{IntM}$ with the interface.

The function $\text{writeInt}()$ is a very simple example of what in embedded software is known as a device driver. It is implemented so that it divides the integer into bytes and iteratively calls the write function $\text{writeByte}()$ of $\text{ByteWriter}$ while ensuring the exclusion specified originally, i.e. no other process can execute the bodies of the functions of $\text{IntWriter}$ implemented by the media in this refinement netlist during this period. $\text{ByteR}$ is the third type, which implements the $\text{IntReader}$ interface and is connected from a port of an object (such as $C0$ in Figure 17.1) that originally accesses $\text{IntM}$ with the interface. As with $\text{ByteW}$, in the implementation of $\text{readInt}()$, the read function $\text{readByte}()$ of $\text{ByteReader}$ (another example of simple driver) is called iteratively to compose an integer.

This refinement netlist is instantiated for each of the original medium objects. The resulting netlist is depicted on the right-end side of Figure 17.5. Note that both refinement netlists ($\text{RefM0}$ and $\text{RefM1}$) are instantiated with the same byte storage $\text{BM}$ in place of $\text{ByteM}$. In this way, the byte storage is shared between the refinement netlists. This is possible for this particular refinement, because it allows the object of $\text{ByteM}$ to be provided externally. While hierarchical refinements are excellent for modularity and re-use, they may be inefficient when it comes to implementation.
17.2 The Metropolis Meta-Model

17.2.5 Coordination constraints

The specification given in Figure 17.1 has two independent data streams. However, its refinement illustrated in Figure 17.5-(b) uses a single object denoted by BM as the storage of both streams. This requires coordination between the two streams. For example, data written by the process P0 must be read by the process C0 rather than C1. For example, in the constructor of the netlist DoubleStream shown in Figure 17.5, it is possible to add the following code:

```plaintext
ltl byteMOrder(IntX p, IntX q, IntX r)
    G(end(p, BM.writeByte) ->
        beg(q,BM.readbyte) U end(r,BM.readbyte);

constraint {
    ltl byteMOrder(P0, C1, C0);
    ltl byteMOrder(P1, C1, C0);
    ltl byteMOrder(P2, C0, C1);
    ltl byteMOrder(P3, C0, C1);
}
```

Here, ltl is a meta-model keyword to specify an LTL formula, where the formula may be optionally preceded by the name of the formula with arguments as shown in this example. In the formula, keywords `beg` and `end` are used to designate events for the beginning and the end of an action respectively.

One can further use variables within the scope of each event as a term of the formula. Inside the keywords, one specifies the name of a process that takes the action and a piece of code; the latter is made of the name of an object in which the code resides and either a label or the name of a function. For instance, `end(p, BM.writeByte)` denotes the end of the execution by the process p of the `writeByte` function defined in the object BM. If there is more than one point in the code at which this function is called by the
process, then the formula applies at all such points. The four instantiated formulas specify the desired coordination.

17.2.6 Quantity constraints

To specify performance and cost constraints in the meta-model, a quantity, such as time, power, or quality of service (QoS), must be defined first. This definition consists of three parts.

The first is the domain of the quantity. For the case of the global time, one may define real numbers as its domain.

The second is a correspondence between the quantity and actions in the behavior. This is the key in this mechanism, since constraints become meaningful only if the correspondence between the system behavior and the quantity is well defined. To establish the correspondence, the meta-model considers only the beginning and the end of an action, which are referred to as events. For example, in the DoubleStream netlist, if we consider as an action the execution of the statement labeled with Wr in Figure 17.2 by the process P0, the beginning and the end of the execution are the events of this action. We then consider the i-th occurrence of an event e in a given execution of the behavior specified in a netlist. We denote it by e[i], where i is a natural number. For example, \( \text{beg}(P0, P0.Wr)[i] \) denotes the beginning of the i-th execution of the statement Wr by P0 that takes place in executing the behavior of the DoubleStream netlist. The correspondence between the quantity and actions is established by defining a function \( f(e[i]) \), called annotation function, that designates the quantity associated with \( e[i] \).

The third part is a set of axioms of the quantity. For the case of global time, if one event follows another event in a given execution, the global time of the former must be no greater than that of the latter.
17.3 Discussion

We summarized the key features of the METROPOLIS environment for the design of complex electronic systems. The environment is based on an infrastructure composed of a meta-model with formal semantics that can be used to capture designs from specification languages and to support simulation, formal analysis and synthesis tools. The framework is conceived to encompass different application domains. Each application domain needs different models, tools and flows. Heterogeneity is supported at the utmost level in METROPOLIS to allow this kind of customization. The meta-model structure has been designed with orthogonalization of concerns in mind: function and architecture, communication and computation are clearly separated. The formal semantics of the meta-model allows embedding of models of computation in a rigorous framework thus favoring design reuse and design chain support. The features of the system should facilitate the dialog among designers with different knowledge domains. The view is not to impose a language or a flow on designers, but rather to make their preferred approach more robust and rigorous. METROPOLIS also offers a set of analysis and synthesis tools that are examples of how the framework can be used to integrate flows.

There is a plan to add tools as different application domains are addressed. At this time, the METROPOLIS team is exploring the automotive, wireless communication and video application domains in collaboration with industrial partners. As what are the critical parts of the design and what needs to be supported to facilitate design hand-offs are understood better, the METROPOLIS team plans to tune the meta-model and to increase the power of the infrastructure for the support of successive refinement as a major productivity enhancement.

METROPOLIS and its components have been made open domain to expose the ideas to the academic and industrial community.
Chapter 18

Proposal for a new interchange format

This chapter is an initial proposal for a potential new interchange format for hybrid systems. The proposed interchange format is by no means complete and it is here explained only to stress that semantics should be favored over syntax in order for an interchange format to be general, well defined and unambiguous.

18.1 Features to be included in the language

An interchange format should be able to capture all the main features of languages that have been already developed. It has to be a sort of maximum common denominator among all hybrid system modeling environments. An interchange format is not supposed to be written by humans but rather by tools that translate the original description to the interchange format. The set of supported features has to be rich enough to guarantee a lossless translation form one language to another. For instance, if the interchange format did not support hierarchy, only flat designs could be described. A translation from one language that supports hierarchy to the interchange
18.1 Features to be included in the language

format would still be possible but it would inevitably flatten out the design structure, making it impossible to retrieve the original description (the translation process then would be lossy in the sense that the design structure would be lost forever).

We describe the set of features that we believe are essential for an interchange format.

- **Object orientation** is used to group common properties of a set of objects in a common base class. It includes the features for defining complex data structures as well as incompletely specified processes. It is possible to extend processes and add/determine part of their behavior. **MODELICA** is a nice example of object oriented programming and highlights the advantages of having such a feature.

- **Hierarchy** is an essential feature for organizing, structuring and encapsulating designs. Flat designs are too complicated to handle because they expose all their complexity in a single view. Even if the interchange format is not supposed to be manipulated directly by designers, it has to keep trace of the original structure. If the entry language supports hierarchy then the interchange format must keep the information in order to allow an inverse transformation.

- **Heterogeneous modeling** is the ability to represent and mix different models of computation. This is one of the main goal of Metropolis and Ptolemy. A hybrid system is the composition of heterogeneous models of computation like continuous time and discrete time. We believe that this feature is extremely important for composing designs coming from different environments.

- **Refinement** is a language feature for specifying a formal relation between components described at different levels of abstraction. Similarly to Ptolemy, refinement can be used to associate a continuous time dynamic to a discrete state or a complex computation to a finite
Proposal for a new interchange format

state transition. Since a design can be expressed at different levels of abstraction, formal refinement is definitely an important feature.

- **Implicit equations** are naturally used by designer to describe dynamical systems. Examples in section 4.3 using Modelica show the effectiveness of using implicit equations in simplifying the design entry. An equation represents a constraint on the variables involved.

- **Explicit declaration of discrete states and transitions manager.** The discrete dynamics is represented as a finite state machine. Each state is characterized by input transitions, output transitions and invariants. Also, each state can be refined into a dynamical system or into another hybrid system (this is a recursive definition). A state is then a place-holder with associated invariant constraints. Transitions from one state to another are guarded by conditions on continuous and discrete state variables. A state machine is described by a set of states and a transition manager that determines the current discrete state. Languages like Ptolemy support the explicit declaration of the finite state machine but the transitions form one state to another, even if defined by the designer, are handled by the simulator.

- **Explicit declaration of invariant constraints.** Invariants are constraints on the state variables. A set of invariant constraints is associated with each state of the finite state machine. A specific logic should be supported by the interchange format to implement this feature. Metropolis, for instance, defines the logic of constraints (LOC) as a general way of declaring relations among quantities.

- **Explicit non-determinism** must be supported by the interchange format. Languages like the Metropolis metamodel have a specific keyword to specify non-deterministic variables. It is up to the simulation engine to implement non-deterministic choices and return, for instance, one of the possible simulation traces. At the specification
level the semantics of a non-deterministic system should include all admissible traces.

- **Explicit declaration of causality relations and scheduling for variables resolution.** Implicit equations support non-causal modeling. The behavior of components is described as a set of equations on variables. Most of the languages for hybrid systems do not support this kind of modeling but they distinguish between input variables (independent) and output variables whose values are computed starting from the input variables. The interchange format has to provide a way to specify causality among variables of a dynamical system.

  Also, given a set of components whose equations include differential and algebraic operators, the order of evaluation of those equations has to be specified. In the case where more that one iteration among all equations is needed (e.g., fix-point semantics), a stop criteria has also to be specified.

- **General continuous/discrete interface.** Each modeling environment defines its own communication semantics between continuous and discrete domains. Instead of defining a communication semantics, the interchange format should provide a set of language primitives that allow the designer to implement any possible communication schema. The Metropolis metamodel, following the principle of orthogonalization of computation and communication, defines a communication medium as an object that offers services, like access to variables. Special synchronization statements are provided by the language to avoid data corruption.

### 18.2 Language syntax

This section describes a pseudo language used in the last section of this chapter to define the semantics of a program.
Proposal for a new interchange format

Rather than focusing on object orientation and scoping, we focus more on the definition of a few base classes and synchronization statements that should be provided by the interchange format. In order to support heterogeneous modeling, the interchange format should provide a set of basic building blocks with which several models of computation can be built. The Metropolis metamodel, for instance, provides three basic components: processes for doing computation, media for communication and quantity managers for synchronization and scheduling.

We list a set of essential objects for describing computation and synchronization:

- **State** is a process that extends the basic process class. It contains ports representing input and output transitions. These ports are connected to other states and are used to communicate output actions and reset maps.

- **AnalogProcess** is a continuous time process which also extends the basic process class. It contains: ports to access external variables that are stored in communication media, and a set of equation statements that define the process behavior.

- **TManager** is the transitions manager which implements the transitions logic of the finite state machine. It defines a resolve method which decides the current state.

- **EquationQM** is the equation quantity manager. Each equation has a scheduler associated with it. The scheduler defines a method resolve that computes unknown values starting from known ones. It uses causality constraints to determine inputs and outputs.

- **EquationResolutionQM** is a quantity manager associated with each dynamical system. It defines a method resolve that implements the algorithm to schedule the equation evaluation.
Also all quantity managers define two other methods: **stable** returns true if the values of the variables are stable (to be defined by the designer depending on the semantics) and **postcond** that takes actions after a stable situation has been reached.

Besides the basic elements, a few keywords are needed:

- **refine**(Object, Netlist) creates a formal relation between an object and a netlist of components. It is used to build models at different levels of abstraction.

- **invariant**\{ <LOC formula> \} is used to specify invariants. All variables used in the formula must be in the scope of the statement.

- **causality**(P, var\(_1\) \(\rightarrow\) var\(_2\)) states that var\(_1\) depends on var\(_2\). The two variables must be in the scope of the process P.

- **scheduling**(P\(_1\), P\(_2\), . . . , P\(_N\)) specifies the scheduling order among processes P\(_1\), . . . , P\(_N\).

### 18.3 Language semantics

Figure 18.1 shows a simple example of hybrid system described in the interchange format. The system has three discrete states which communicate through media. Each state is refined into a dynamical system (or into another hybrid system). State Mode3, for instance, is refined into a dynamical system composed of two continuous processes, A1 and A3. A continuous process is described using equations.

A graphical representation helps in understanding the language semantics: squares represent processes, circles represent communication media and finally diamonds represent quantity managers.

The scheduled netlist contains processes which generate events. The scheduling netlist limits the possible events that processes can execute by imposing scheduling constraints.
Figure 18.1: Graphical representation of an hybrid system using the interchange format.
18.3 Language semantics

The execution of a program is defined as sequences of event vectors $v$ whose size is the number of processes in the scheduled netlist. The execution of a process $P_i$ is defined as the sequence of events $\{v[i]\}$. At each state of the entire program, processes issue requests to execute a set of possible events to the scheduling netlist which decides the event vector that satisfies the scheduling constraints. If more than one event vector complies with the constraints, the choice among them is non-deterministic.

The type of request to the scheduling netlist depends on the process that issues it. State processes issue requests to execute their output transitions. These events can be executed only if guard conditions are satisfied and/or invariant constraints are violated. AnalogProcess processes issue requests to evaluate their equations sets. These events can be executed only if the dynamical system is a refinement of the current state. Also the equations have to be evaluated according to the order and causality specified by the scheduling and causality constraints.

A netlist execution starts with the request by the initial state to the transitions quantity manager $TQM$, of executing its output transitions. $TQM$ starts a resolution algorithm which checks all guards and invariant constraints. If all guards are false and all invariants are met, the transitions quantity manager calls the equation resolution quantity manager $ERQM$ of the current state refinement.

$ERQM$ has to determine a value for all continuous variables defined in the dynamical system it is associated with. The resolution algorithm is described in the resolve function and takes into account scheduling constraints. The algorithm could run for more than one iteration depending on the continuous time domain semantics. The resolve function will invoke the resolve functions defined in the equation quantity mangers which use the causality constraint for determining outputs starting from inputs.

When the equation resolution algorithm terminates, the transitions quantity manager takes over and checks guards and invariants again. If all invari-
Figure 18.2: Simple hybrid system example. A) is the schematic representation of the circuit, B) shows the finite state machine, transitions and invariants.

Ants are met and if no guard event was missed, transition quantity manager exits the resolution phase without changing the current state and continuous variables are annotated with their values. Finally, control goes back to the scheduled netlist which issues another set of requests.

18.4 Examples

Consider the continuous time system of figure 18.2. Resistor $R$ and capacitor $C$ are two continuous time actors. A capacitor could be described, for instance, in the following way:

**Capacitor** is a process derived from a general analog process. **AnalogProcess** base class defines, for instance, special functions for connecting itself to quantity managers. The process has two ports to connect to communication media and read/write variables value. The port type is an interface which declares services that are defined (implemented) by communication media. Note that ports are not associated with a direction which implies that the component does not have a causality constraint associated with its description.

A resistor is described in the same way but the current/voltage relation
18.4 Examples

process Capacitor extends AnalogProcess {
    parameter double C;
    port AnalogInterface i;
    port AnalogInterface v;
    equations {
        i = c * der(v);
    }
}

Figure 18.3: Example of code for describing a capacitor

is governed by Ohm’s law \( v = R \cdot i \).

The entire continuous time subsystem results from the interconnection of analog processes into a netlist. Figure 18.4 shows how a continuous time netlist is specified using the interchange format. Causality constraints and scheduling constraints are specified in this netlist and are used to build the scheduling netlist.

An example of state is shown in figure 18.5.

Reset maps as well as shared state variables are all accessed through ports. A media has to provide a place to store these variables and also has to implement a services to access them. Depending on the implementation of those services, it is possible to customize the communication semantics.

A finite state machine is represented as interconnection of states and transitions in figure 18.6.

The first part of the netlist instantiates all components including states and communication media. The second part connects states to channels. The last part describes the transitions. Each state declares a set of output transitions that can be connected to the target state in the FSM netlist.

A top netlist is needed for instantiation of the finite state machine and association of dynamical systems to states. A snippet of the code is shown in figure 18.7.
netlist RCCircuit extends AnalogNetlist {
    port AnalogInterface V0;
    AnalogChannel current = new AnalogChannel(0.0);
    AnalogChannel voltagec = new AnalogChannel(0.0);
    AnalogChannel voltager = new AnalogChannel(0.0);
    Sub S = new Sub();
    Capacitor C = new Capacitor(1uF);
    Resistor R = new Resistor(1K);
    connect(S.in1,V0);
    connect(S.in2,voltagec);
    connect(S.out,voltager);
    connect(R.v,voltager);
    connect(R.i,current);
    connect(C.i,current);
    connect(C.v,voltagec);
    constraints{
        causality(R,v->i);
        causality(C,i->v);
        causality(S,out-> in1 && in2);
        scheduling(S->R->C);
    }
}

Figure 18.4: Example of code describing an analog netlist.

The top netlist uses the refine keyword to associate a dynamical system to a state. A few more connections are carried out in the top netlist. First of all we have to connect the reset maps to the dynamical system input. In this case the variable $V_0$ is an input of the RCCircuit netlist. Also we have to connect the variable corresponding to voltage across the capacitor to the state input port. This variable will be checked during simulation for
18.4 Examples

process Charge extends State{
    port AnalogChannel v0out;
    port AnalogChannel v0in;
    port AnalogChannel vc;
    OutTransition vcth(vc >= 4, v0out = -5);
    constraints{
        invariant(vc>=1 && vc <= 4 && der(vc) >= 0);
    }
}

Figure 18.5: Example of code describing an state

netlist RCFSM extends FSMNetlist{
    Charge ch = new Charge();
    Discharge dch = new Discharge();
    AnalogChannel v0c2d = new AnalogChannel(0.0);
    AnalogChannel v0d2c = new AnalogChannel(0.0);
    connect(ch.v0out,v0c2d);
    connect(ch.v0in,v0d2c);
    connect(dch.v0out,v0d2c);
    connect(dch.v0in,v0c2d);
    transition(ch.vcth,dch)
    transition(dch.vcth,ch);
}

Figure 18.6: Example of code describing an FSM.

evaluating guards conditions and invariant constraints.
Proposal for a new interchange format

netlist Top {
    RCFSM myfsm = new RCFSM();
    refine(myfsm.ch,RCCircuit);
    refine(myfsm.dch,RCCircuit);
    refineconnect(myfsm.ch.v0in,refinementof(myfsm.ch).V0);
    refineconnect(myfsm.dch.v0in,refinementof(myfsm.dch).V0);
    connect(myfsm.ch.vc,refinementof(myfsm.ch).voltagec);
    connect(myfsm.dch.vc,refinementof(myfsm.dch).voltagec);
}

Figure 18.7: Example of code describing the top netlist.

18.4.1 Cut-off Control

In this example, extracted from [19], we model an automobile engine and power train as a hybrid system in order to design a controller for its cut-off region. The behaviors of an automobile engine are divided into regions of operation, each characterized by appropriate control actions to achieve a desired result. The cut-off region is entered when the driver releases the accelerator pedal, thereby requesting that no torque be generated by the engine. In order to minimize power train oscillations that result from suddenly reducing torque, a closed loop control damps the oscillations using carefully timed injections of fuel. The control problem is therefore hybrid, consisting of a discrete (the fuel injection and the cylinders) and a continuous (the power train behavior) systems tightly linked.

Figure 18.8 and Figure 18.9 show a schematic representation, and the corresponding code, of a cylinder that goes through the four phases of intake (I), compression (C), expansion (E) and exhaust (H). The transitions of the FSM occur when the piston reaches the bottom or top dead point. The guard condition enabling the transition is expressed in terms of the piston position $\phi$ measured on the crankshaft. The snippet of code shown
18.4 Examples

in Figure 18.10 instantiates all the states and defines the transition relation as seen in the previous example.

Figure 18.8: Cylinder schematics

The power-train model, shown in Figure 18.11, is a linear fifth order model, a reduced order model of power-train dynamics describing the important phenomena involved in power-train oscillations of interest in cut-off control. The matrices $a$ and $b$ that describe the dynamics must be provided as parameters to the model (see [20] for more details). The state includes the engine block angle $\alpha_b$, the wheel revolution speed $\omega_p$, the axle torsion angle $\alpha_e$, the crankshaft revolution speed $\omega_c$, and the crankshaft angular position $\phi_c$. The input signal $u$ is the torque acting on the crank. Such signal is a complex function of time during the expansion phase that depends on the dynamics of the mix explosion. We assume for $u$ a piece-wise constant model, set to zero during the $I$, $C$ and $H$ phases, and to an average value during the $E$ phase. The position of the individual cylinders mounted on the crankshaft is obtained by adding an appropriate offset to the crankshaft
Proposal for a new interchange format

process Intake extends State {
    port AnalogChannel phit;
    port StateVariable z;
    port StateVariable u;
    port AnalogChannel r;

    OutTransition phith( phit == 180, z = z * r, u = 0 );
}

process Expansion extends State {
    parameter double G;

    port AnalogChannel phit;
    port StateVariable z;
    port StateVariable u;
    port AnalogChannel q;
    port BinaryChannel j;

    OutTransition phith( phit == 180, z = G * q * j, u = 0 );
}
...

Figure 18.9: Cylinder states

position, using the module shown in Figure 18.12. This module is used in the code for the complete engine, shown in Figure 18.13, that connects four cylinders and computes the total torque by combining the contributions of
18.4 Examples

each cylinders. This is accomplished by the module shown in Figure 18.14.

A controller can be again represented as an analog process that produces
the appropriate injection signals to the cylinders. Different controllers are
possible. In particular, the designer might initially describe the behavior as
a continuous time process, and then refine the solution to a discrete time
process that can be implemented on a digital processor. This is, for example,
the design processes described in [19].
Proposal for a new interchange format

netlist Cylinder extends FSMNetlist {
    // mass of air-fuel mix
    port AnalogChannel q;
    // fuel present/absent in the mix
    port BinaryChannel j;
    // modulation factor for spark advance
    port AnalogChannel r;
    // Crankshaft position
    port AnalogChannel phit;

    StateVariable z = new StateVariable(0);
    StateVariable u = new StateVariable(0);
    Intake in = new Intake();
    Compression co = new Compression();
    Expansion ex = new Expansion();
    Exhaust eh = new Exhaust();

    connect( in.phit, phit ); connect( co.phit, phit );
    connect( ex.phit, phit ); connect( eh.phit, phit );
    connect( in.z, z ); connect( co.z, z );
    connect( ex.z, z ); connect( eh.z, z );
    connect( in.u, u ); connect( co.u, u );
    connect( ex.u, u ); connect( eh.u, u );
    connect( in.r, r ); connect( ex.q, q );
    connect( ex.j, j );
    transition( in.phith, co );
    transition( co.phith, ex );
    transition( ex.phith, eh );
    transition( eh.phith, in );
}

Figure 18.10: Example code for complete cylinder.
18.4 Examples

```java
process powerTrain extends AnalogProcess {
    // Input torque
    port u;
    // Engine block angle
    port AnalogChannel alphab_b;
    // Wheel revolution speed
    port AnalogChannel omega_p;
    // Axle torsion angle
    port AnalogChannel alpha_e;
    // Crankshaft revolution speed
    port AnalogChannel omega_c;
    // Output crankshaft angular position
    port AnalogChannel phi_c;
    // powerTrain linear dynamics
    parameter a[4][4];
    // powerTrain dependency on torque
    parameter b[4];

    equations {
        alphab_b = a[1,1]*alphab_b + a[1,2]*omega_p + a[1,3]*alpha_e +
                   a[1,4]*omega_c + b[1]*u;
        omega_p = a[2,1]*alphab_b + a[2,2]*omega_p + a[2,3]*alpha_e +
                   a[2,4]*omega_c + b[2]*u;
        alpha_e = a[3,1]*alphab_b + a[3,2]*omega_p + a[3,3]*alpha_e +
                   a[3,4]*omega_c + b[3]*u;
        omega_c = a[4,1]*alphab_b + a[4,2]*omega_p + a[4,3]*alpha_e +
                   a[4,4]*omega_c + b[4]*u;
        phi_c = 6 * omega_c;
    }
}
```

Figure 18.11: Example code for power-train.
Proposal for a new interchange format

process computeOffset extends AnalogProcess {

    port AnalogInterface in1;
    port AnalogInterface in2;
    port AnalogInterface out;

    double sum;

    equations {
        sum = ( in1 + in2 ) % 360;
        out = sum <= 180 ? sum : 360 - sum;
    }

}

Figure 18.12: Code for offset computation.
18.4 Examples

netlist Engine {

    port AnalogChannel torque;
    port AnalogChannel q;
    port BinaryChannel j;
    port AnalogChannel r;

    port AnalogChannel phi;

    double phi1, phi2, phi3, phi4;
    double u1, u2, u3, u4;

    computeOffset( phi, 0, phi1 );
    computeOffset( phi, 180, phi2 );
    computeOffset( phi, 0, phi3 );
    computeOffset( phi, 180, phi4 );

    Cylinder c1 = new Cylinder( q, j, r, phi1 );
    Cylinder c2 = new Cylinder( q, j, r, phi2 );
    Cylinder c3 = new Cylinder( q, j, r, phi3 );
    Cylinder c4 = new Cylinder( q, j, r, phi4 );

    connect( c1.u, u1 );
    connect( c2.u, u2 );
    connect( c3.u, u3 );
    connect( c4.u, u4 );

    combineTorque( u1, u2, u3, u4, torque );

}

Figure 18.13: Code for a 4-piston engine
process combineTorque extends AnalogProcess {

    port AnalogInterface u1;
    port AnalogInterface u2;
    port AnalogInterface u3;
    port AnalogInterface u4;

    port AnalogInterface u;

    equations {

        u = u1 + u2 + u3 + u4;

    }

}
Chapter 19

Concluding Remarks

In this chapter, we give a comparative summary of the design approaches, languages, and tools discussed throughout this report and then we make some considerations on the important topic of developing a standard interchange format for hybrid system design.

19.1 Comparative Summary

Table 19.1 summarizes the distinctive features of the various modeling and design environments, programming languages, simulators and tools for hybrid systems that we have discussed in the previous chapters.

Table 19.2 shows the approach adopted by each language for modeling the basic hybrid system structure. The first column indicates how discrete and continuous signals are declared in each language. Some languages like CHARON and MODELICA use special type modifiers to specify whether a variable is discrete or continuous. However, the semantics is different in the two cases. While CHARON defines a discrete variable constant between two events, hence having derivative equal to zero, the derivative of discrete variables in MODELICA is not defined. Graphical languages like HyVISION, SIMULINK, and SCICOS rely on attributes associated with ports. Also, signal


## Table 19.1: Nature and main features of the various modeling approaches’ toolsets.

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature</th>
<th>Main Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charon</td>
<td>modeling language</td>
<td>formal semantics for hierarchy, concurrency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>simulator, type checker, interface to Java</td>
</tr>
<tr>
<td>CheckMate</td>
<td>verification toolbox</td>
<td>formal semantics (TEDHS) for simulation and verification</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integrated with Matlab Simulink/Stateflow</td>
</tr>
<tr>
<td>HSIF</td>
<td>interchange format</td>
<td>modeling of networks of hybrid automata</td>
</tr>
<tr>
<td></td>
<td></td>
<td>simulation through HyVISUAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integrated with Matlab Simulink/Stateflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>formal semantics for hierarchy, concurrency, refinement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>simulator, type checker, interface to Java</td>
</tr>
<tr>
<td>Hysdel</td>
<td>modeling language</td>
<td>modeling of discrete-time affine dynamical systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>generation of input for Matlab simulation</td>
</tr>
<tr>
<td>HyVisual</td>
<td>visual model</td>
<td>modeling and simulation of hybrid systems, hierarchy support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ptolemy II-based block-diagram editor</td>
</tr>
<tr>
<td>Masaccio</td>
<td>formal model</td>
<td>support for concurrent, sequential, and timed compositionality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enables assume-guarantee reasoning</td>
</tr>
<tr>
<td>Metropolis</td>
<td>design environment</td>
<td>support for heterogeneous design, formal refinement, mapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>logic of constraints for quantitative verification</td>
</tr>
<tr>
<td>Modelica</td>
<td>modeling language</td>
<td>object-oriented modeling of heterogeneous physical systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modelica standard library, commercial tools</td>
</tr>
<tr>
<td>Scicos</td>
<td>hybrid system toolbox</td>
<td>modeling and simulation of hybrid systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C code generation, interface to SynDEx</td>
</tr>
<tr>
<td>Shift</td>
<td>programming language</td>
<td>modeling of dynamic networks of hybrid automata</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C code generation, λ-Shift for real-time control</td>
</tr>
<tr>
<td>Sildex</td>
<td>integrated toolset</td>
<td>formal semantics (a la Signal), embedded code generation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>simulator, formal proofs, interface to Simulink</td>
</tr>
<tr>
<td>Simulink</td>
<td>interactive tool</td>
<td>analysis and simulation, hierarchy support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>chart animation, library of predefined blocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integrated with Matlab Simulink/Stateflow</td>
</tr>
<tr>
<td>Stateflow</td>
<td>interactive tool</td>
<td>statechart formalism, hierarchy support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>chart animation, library of predefined blocks</td>
</tr>
<tr>
<td>SynDex</td>
<td>system-level CAD</td>
<td>real-time code generation, distribution and scheduling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Petri net, support for hardware models</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Concluding Remarks**
19.1 Comparative Summary

types can be automatically inferred during compilation by analyzing statically the system topology. HYSDEL and CHECKMATE describe the hybrid system as a finite state machine connected to a set of dynamical systems. This makes the separation of discrete and continuous signals very clear.

Another basic feature is the association of a dynamical system to a specific state of the hybrid automata. HYVISUAL and CHARON seem to have the most intuitive syntax and semantics for this purpose. In HYVISUAL a state of the hybrid automata can be refined into a continuous time system. CHARON allows a mode to be described by a set of algebraic and differential equations. In CHECKMATE, SIMULINK, and HYSDEL an hybrid system is modeled as two main blocks: a state machine and a set of dynamical systems. The automata is described by a finite state machine with state and transitions on the arcs. A transition can be triggered by an event coming from a particular event-generation block that monitors the values of the variables of the dynamical system. On the other hand, the finite state machine can generate events that are sent to a mode-change block whose purpose is to select a particular dynamics depending on the events. SCICOS implements the automata directly as an interconnection of blocks whose events can affect the continuous state of blocks implementing the continuous dynamics. Finally, MODELICA provides a set of statements that can add equations to the systems depending on the events.

The last column in Table 19.2 describes how discrete and continuous signals and blocks interact with each other. CHECKMATE and HYSDEL use an event-generator and a mode-change block. HYVISUAL and SIMULINK provide special library blocks to convert a discrete signal into a continuous and vice-versa. In SCICOS, a block can have both continuous and discrete inputs as well as continuous and discrete states. Discrete states can influence continuous states. CHARON and MODELICA have special modifiers for distinguishing between discrete and continuous signals. As in all other languages, assignments of one to the other is not possible and can be statically
### Concluding Remarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Continuous/Discrete Specification</th>
<th>State/Dynamics Mapping</th>
<th>Continuous/Discrete Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Charon</strong></td>
<td>defined by language modifier</td>
<td>modes refinement into continuous dynamics</td>
<td>indirect</td>
</tr>
<tr>
<td><strong>CheckMate</strong></td>
<td>separation between FSMs and dynamical systems</td>
<td>discrete output from FSMs to dynamical systems</td>
<td>event generator first order hold</td>
</tr>
<tr>
<td><strong>HySDEL</strong></td>
<td>real and boolean signals</td>
<td>discrete output from FSMs to dynamical systems</td>
<td>event generator first order hold</td>
</tr>
<tr>
<td><strong>HyVisual</strong></td>
<td>signal attribute, automatic type detection</td>
<td>state refinement into continuous models</td>
<td>toContinuous, toDiscrete actors</td>
</tr>
<tr>
<td><strong>Modelica</strong></td>
<td>defined by language modifier</td>
<td>different equation sets depending on events</td>
<td>indirect (when statements)</td>
</tr>
<tr>
<td><strong>Scicos</strong></td>
<td>defined by port attribute</td>
<td>implemented by connections of event selectors</td>
<td>interaction between continuous/discrete states</td>
</tr>
<tr>
<td><strong>Simulink</strong></td>
<td>automatic type detection</td>
<td>discrete output from FSMs to dynamical systems</td>
<td>library blocks like zero-order hold</td>
</tr>
</tbody>
</table>

Table 19.2:

checked (by a simple type checker).

Table 19.3 shows the features provided by the different tools. All of them support the derivative operator. Specification of the discrete automata has different interpretations. Again, the most intuitive way of describing the discrete automata is implemented by HyVisual and Charon. HyVisual has a finite state machine editor where a state machine can be described with bubbles and arcs. Each bubble can then be refined into a continuous time system or into another hybrid system. Also Charon has graphical environment where modes can be instanced and connected by arcs. HYSDEL specifies the finite state machine with boolean formulas.

An interesting and useful feature is object orientation (OO). By object orientation we mean the possibility of defining object and extending them through inheritance and field/method extension. From this point of view, SIMULINK is not object oriented since it is not possible to define a subsystem and then inherit its properties and add other capabilities.

Another very important feature is the possibility of modeling non-causal systems. MODELICA is the only language that allows non-causal modeling.

None of the languages considered in this report has a clear definition of
19.2 Towards the Development of a Standard Interchange Format

An interchange format is a file, or a set of files, that contains data in a given syntax that is understood by different interacting tools. It is not a database nor a data structure, but a simpler object whose goal is to foster the exchange of data among different tools and research groups.

The final goal of this project is to recommend a standard for an interchange format among tools that deal with hybrid systems thus creating a fertile ground for further growth of the field and for the pervasive use of hybrid technology in industry. In the U.S., the DARPA MoBIES project had the importance of an interchange format very clear and supported the devel-

### Table 19.3:

<table>
<thead>
<tr>
<th>Name</th>
<th>Derivative</th>
<th>Automata Definition</th>
<th>Hierarchy</th>
<th>Object</th>
<th>Non-Causal Modeling</th>
<th>Algebraic Loops</th>
<th>Dirac Pulses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charon</td>
<td>yes</td>
<td>modes of operation</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CheckMate</td>
<td>yes</td>
<td>Stateflow specification</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>HyVedel</td>
<td>discrete</td>
<td>logic functions</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>HyVisual</td>
<td>integration</td>
<td>graphical editor</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Modelica</td>
<td>yes</td>
<td>algorithm sections</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>not yet</td>
</tr>
<tr>
<td>Scicos</td>
<td>integration</td>
<td>network of condit. blocks</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Simulink</td>
<td>derivative and integration</td>
<td>Stateflow specification</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

the semantics of programs that contain algebraic loops. All of them rely on the simulation engine that cannot solve algebraic loops and will stop with an error message. We believe that a language has to give a meaning to programs containing algebraic loops and the meaning should be independent from the simulator’s engine.
opment of HSIF as a way of fostering interactions among its participants. As such, HSIF is the obvious candidate for recommendation. However, we believe that some actions must be taken to adjust HSIF or to adopt another approach since some limitations to its semantics make the interchange of data between foreign tools difficult (for example, HSIF does not support some of the features of Simulink/Stateflow model). To motivate our views, we offer here some considerations about interchange formats that are the result of experience in the field of Electronic Design Automation and of a long history in participating to the formation of standard languages and models for hardware design.

We list here what we believe are fundamental characteristics of any interchange format for tools and designs: An interchange format must:

- support all existing tools, modeling approaches and languages in a coherent global view of the applications and of the theory;
- be open, i.e., be available to the entire community at no cost and with full documentation;
- support a variety of export and import mechanisms;
- support hierarchy and object orientation (compact representation, entry error prevention).

By having these properties, an interchange format can become the formal backbone for the development of sound design methodologies through the assembly of various tools. In general, a design automation flow is made of tools that have different purposes: specification, simulation, synthesis, formal verification, ... Hence, they are often based on different formalisms and operate on the design at different levels of abstraction. The role of the interchange format is to facilitate the translation of design specification from one tool to the other. As illustrated in Figure 19.1, the process of moving from the design representation used by tool $A$ to the one used by
Towards the Development of a Standard Interchange Format

Figure 19.1: Role of an interchange format for design tools.

Tool A

Tool B

Translating

Preprocessing

Standard Interchange Format

tool \( B \) is structured in two steps: first, a representation in the standard interchange format is derived from the design entry that is used by \( A \), then a preprocessing step is applied to produce the design entry on which \( B \) can operate. Notice that tool \( B \) may not need all the information on the design that were used by \( A \) and, as it operates on the design, it may very well produce new data that will be written into the interchange format but that will not ever be used by \( A \). Naturally, the semantics of the interchange format must be rich enough to capture and “protect” the different properties of the design at the various stages of the design process. This guarantees that there will be no loss going from one design environment to another due to the interchange format itself. The format is indeed a neutral go-between.

HSIF does not support some of the models of important tools and it does not allow hierarchical representations. In our opinion, HSIF is an excellent model for supporting clean design of hybrid systems but not yet a true interchange format. **Simulink/Stateflow** internal format could be
Concluding Remarks

a de facto standard but it is not open nor it has features that favor easy import and export. MODELICA has full support of hierarchy and of general semantics that subsumes most if not all existing languages and tools. As such, it is indeed an excellent candidate but it is not open. In addition, all of them have not been developed with the goal of supporting heterogeneous implementations.

On the other hand, the Metropolis meta-model (MMM) has generality and can be used to represent a very wide class of models of computation. It has a clear separation between communication and computation as well as architecture and function. However, we have limited experience (if any) with the use of the meta-model to represent hybrid system control. We have plans to extend the environment and the model to cover adequately continuous time systems. The meta-model itself is perfectly capable to express continuous time systems. However, there is no tool that can manage at this time this information in Metropolis.

In conclusion, we believe that a full fledged recommendation at this time is premature given the state of maturity of the available approaches. We believe that combining and leveraging HSIF, MODELICA, and the Metropolis meta-model, we can push for the foundations of a standard interchange format as well as a standard design capture language where semantics is favored over syntax.
Bibliography


BIBLIOGRAPHY


[87] Specification and Description Language SDL, 1988. 113

[88] Estelle - A formal description technique based on extended state transition model. 113


BIBLIOGRAPHY


0207. INRIA, (Rocquencourt, France, June), 1997. 13, 67, 70, 71, 73, 74, 75


BIBLIOGRAPHY


