



## Project IST-2001-38314 COLUMBUS

Design of Embedded Controllers for Safety Critical Systems

### WPHS: Hybrid System Modeling

# Overview of methods representing semantics of hybrid modeling techniques

**Luca Berardi<sup>1</sup>, Maria D. Di Benedetto<sup>1</sup> and  
Alberto Sangiovanni-Vincentelli<sup>2</sup>**

***August 5, 2003***

***Version: 0.2***

***Task number:***

***Deliverable number: DHS2***

---

<sup>1</sup> University of L'Aquila

<sup>2</sup> PARADES and University of California at Berkeley

**Contract:** IST-2001-38314 of European Commission

## DOCUMENT CONTROL SHEET

**Title of document:** Overview of methods representing semantics of hybrid modeling techniques

**Authors of document:** Luca Berardi, Maria D. Di Benedetto and Alberto Sangiovanni-Vincentelli

**Deliverable number:** DHS2

**Contract:** IST-2001-38314 of European Commission

**Project:** Design of Embedded Controllers for Safety Critical Systems (Columbus)

## DOCUMENT CHANGE LOG

Version #	Issue Date	Sections affected	Relevant information
0.1	30 June 03	All	First draft
0.2	5 August 03	All	Second draft

Author(s) and Reviewers		Organisation	Signature/Date
<b>Authors</b>	L. Berardi	AQUI	
	M.D. Di Benedetto	AQUI	
	A. Sangiovanni - Vincentelli	PARADES & UCB	
<b>Internal reviewers</b>	A. Balluchi	PARADES	
	G. Pola	AQUI	

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>The LSV Tagged-Signal Model</b>	<b>9</b>
2.1	The LSV Model . . . . .	10
2.1.1	Processes . . . . .	11
2.1.2	Concurrency and Communication . . . . .	13
2.2	Basic time. . . . .	13
2.3	Treatment of time in processes. . . . .	14
2.4	Implementation of concurrency and communication. . . . .	15
2.5	Examples of LSV model representation of commonly used models of computation . . . . .	16
2.6	Conclusion . . . . .	17
<b>3</b>	<b>Trace Algebras</b>	<b>18</b>
3.1	Introduction . . . . .	19
3.2	The Trace Algebra Approach . . . . .	20
3.2.1	Traces and Trace algebra operators . . . . .	22
3.2.2	Homomorphisms and Conservative approximations . . . . .	24
3.3	Comparison with other approaches . . . . .	26
<b>4</b>	<b>Behavioral Systems</b>	<b>29</b>
4.1	The behavioral approach . . . . .	29
4.1.1	Input/Output Representation . . . . .	31

---

4.1.2	Modular representation of behavioral systems . . . . .	32
4.1.3	Controllability and Observability . . . . .	33
4.2	Hybrid Systems and the behavioral approach . . . . .	34
4.2.1	Behavioral state representation of hybrid systems . . . . .	35
4.2.2	Concatenability of behaviors in hybrid systems . . . . .	37
4.2.3	Robust hybrid control using the behavioral approach . . . . .	39
4.3	Behavioral Systems and Trace Algebras . . . . .	41
<b>5</b>	<b>The Hybrid System Interchange Format</b>	<b>45</b>
5.1	Introduction . . . . .	46
5.2	Syntax . . . . .	46
5.2.1	Hybrid automaton . . . . .	46
5.2.2	Network of hybrid automata . . . . .	49
5.3	Semantics . . . . .	50
5.3.1	Semantics of a single automaton . . . . .	50
5.3.2	Semantics of the network . . . . .	53
5.4	Notes and discussion . . . . .	54
5.5	Conclusions . . . . .	60
<b>6</b>	<b>The Metropolis Framework</b>	<b>62</b>
6.1	The Metropolis methodology: features and goals . . . . .	63
6.2	The Metropolis Meta-Model . . . . .	65
6.2.1	Behaviors as actions . . . . .	66
6.2.2	Action automata . . . . .	67
6.2.3	Processes and media . . . . .	69
6.2.4	Refinement . . . . .	73
6.2.5	Coordination constraints . . . . .	76
6.2.6	Quantity constraints . . . . .	77
6.3	Conclusions . . . . .	78
<b>7</b>	<b>Conclusions</b>	<b>79</b>

# Chapter 1

## Introduction

*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 workpackage in Columbus.*

The first step in this direction was collecting data and compiling the first Columbus deliverable, DHS1. In that report, we focused on a few available languages, formalism and tools that have been proposed in the past years in the US. In particular, we focused on hybrid-system languages that are the basis for some popular industrial verification tools such as Simulink and State Flow, and for some academic work, CheckMate, Charon and Masaccio. We stressed that the roots of the difficulties in combining tools and environments for hybrid system design lay in the semantics aspects.

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 this second deliverable, DHS2, we focus on this aspect and we review the most recent approaches to the mathematics and the semantics aspect of hybrid system modeling. 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 work 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.

The approach by the MoBIES project with its Hybrid System Interchange Format (HSIF) is to focus on a data model to use for sharing information and transferring this information among different tools. HSIF for its origing is a strong candidate for our final recommendation. However, we believe that some decisions related to restrictions to its semantics may have a negative impact on the appeal to users. 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 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. This and other considerations are interspersed throughout the report. The report forms with DHS1 a rather comprehensive survey of the most influential approaches to hybrid system modeling. However, we believe some more work needs to be done to consider two particular approaches that can provide additional benefit to the outcome of the project: Modelica and the semantics of a tool for hybrid system capture and simulation developed at INRIA. For this reason, we will provide in the next deliverable a description of these other approaches.

This report DHS2 is structured as follows: Chapter 2 describes the Lee and Sangiovanni-Vincentelli (LSV) tagged-signal model, a denotational model for comparing models of computation. Chapter 3 presents a novel approach to modeling concurrency in heterogeneous systems using trace theory.

Chapter 4 describes the behavioral approach to system theory.

Chapter 5 introduces the Hybrid System Interchange Format proposed by Vanderbilt.

Chapter 6 focuses on the Metropolis Metamodel as a potential unifying framework for tools and models.

Finally, Chapter 7 presents future work and some concluding remarks.

## Chapter 2

# The LSV Tagged-Signal Model

A design (at all levels of the abstraction hierarchy from functional specification to final implementation) is generally represented as a set of components, which can be considered as distinct monolithic blocks, interacting with each other and with an environment that is not part of the design. The model of computation (MOC) defines the behavior and interaction of these blocks. Compactness of description, fidelity to design style, ability to synthesize to an appropriate implementation and to optimize its behavior, are criteria that guide the selection of an MOC. For example, some MOCs are suitable for describing complicated data transfer functions and completely unsuitable for complex control, while others are designed with complex control in mind. Hybrid systems in general terms can be viewed as formalisms for describing a complex system using combinations of MOCs when a single one is not powerful, expressive or practical enough.

Traditionally, hybrid systems have been defined as finite automata in which each state is associated with a set of differential equations, and transitions are triggered based on state and input values of the differential equations. However, the interactions among the discrete and the continuous

dynamics are non-trivial. To capture these interactions rigorously and to prevent confusion that has often arisen by their improper modeling, the concept of hybrid system execution has been recently introduced in [24]. We believe that the concept of hybrid system can and should be extended to contain any combination of different models of computation, not necessarily finite state automata and ordinary differential equations. In a “hybrid” model of the engine and of the power-train we used three different models of computation. We used the theory behind the notion of models of computation to identify the role of the interfaces among the different components of the hybrid system.

## 2.1 The LSV Model

The LSV tagged-signal model (LSV model) proposed by Lee and Sangiovanni-Vincentelli [23] 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  $\mathbf{s}$ , 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  $\perp$ , which indicates the

## 2.1 The LSV Model

---

*absence of a value*. It should be noticed that  $(\perp, t) \notin \lambda$ , indeed  $(\perp, t)$  does not satisfy  $(\perp, t) \cup s = s$  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 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.

### 2.1.1 Processes

A *process*  $P$  is a subset of the set of all  $n$ -tuples of signals  $S^n$  for some  $n$ . A particular  $\mathbf{s} \in S^n$  is said to *satisfy* the process if  $\mathbf{s} \in P$ . An  $\mathbf{s}$  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.

Different order relations on the set of tags  $T$ , in the LSV model framework partition the space of process representations. 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 envi-

ronment 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  $\mathbf{s}$  can be written  $\mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2)$ , where  $\mathbf{s}_1 \in S^i$  is an  $i$ -tuple of *input signals* for process  $P$  and  $\mathbf{s}_2 \in S^o$  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  $(\mathbf{s}_1, \mathbf{s}_2) \in F$  and  $(\mathbf{s}_1, \mathbf{s}_3) \in F$ , then  $\mathbf{s}_2 = \mathbf{s}_3$ . In this case, we can write  $\mathbf{s}_2 = F(\mathbf{s}_1)$ , 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., [39, 29]) 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  $\mathbf{s}$ , define  $\mathbf{s}_{>t}$  to be the tuple of the (possibly empty) subset of the events in  $\mathbf{s}$  with tags greater than  $t$ . Two input signal tuples  $\mathbf{r}, \mathbf{s} \in S^i$  are in relation  $E_t^F$  (denoted  $(\mathbf{r}^i, \mathbf{s}^i) \in E_t^F$ ) if  $\mathbf{r}_{>t} = \mathbf{s}_{>t}$  implies  $F(\mathbf{r})_{>t} = F(\mathbf{s})_{>t}$ . This definition intuitively means that process  $F$  cannot distinguish between the “histories” of  $\mathbf{r}$  and  $\mathbf{s}$  prior to time  $t$ . Thus, if the inputs are identical after time  $t$ , then the outputs will also be identical.  $E_t^F$  is obviously an equivalence relation,

## 2.2 Basic time.

---

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  $A$ .

### 2.1.2 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.

## 2.2 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<sup>1</sup>.

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 is synchronous with an event in the other signal and vice versa.

### 2.3 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”.

---

<sup>1</sup>Formally, there exists a function  $d : T \times T \rightarrow \mathbb{R}$  mapping pairs of tags to real numbers such that  $d(t_1, t_2) \geq 0$  where  $d(t_1, t_2) = 0 \Leftrightarrow t_1 = t_2$ ,  $d(t_1, t_2) = d(t_2, t_1)$  and  $d(t_1, t_2) + d(t_2, t_3) \geq d(t_1, t_3)$  for any  $t_1, t_2, t_3$ .

### 2.4 Implementation of 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.

## 2.5 Examples of LSV model representation of commonly used models of computation

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<sup>2</sup>. In particular, we present the categorization of three 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

---

<sup>2</sup>Note that the specification for the controller may use untimed models. In this paper we focus on physical models and control laws and this is the reason for the use of timed models.

## 2.6 Conclusion

---

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.

## 2.6 Conclusion

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 3

# Trace Algebras

This Chapter is based on a denotational view of hybrid systems worked out through the *trace* concept citeBurch01, Burch01b, Burch02. A trace is intuitively a representation of a single instance of the behavior of a generic system<sup>1</sup>.

Under this perspective traces will be 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 un-

---

<sup>1</sup>This holds true not only for hybrid systems, but for every system which superimposes a partial order on the succession of events

### 3.1 Introduction

---

necessary 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. In Chapter 4, we will compare the two approaches in some degree of detail.

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.

### 3.1 Introduction

The concept of trace is not new in the scientific literature dealing with real-time concurrent systems.<sup>2</sup>

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 [18] and [19]; 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 [6], [7] and [34].

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

---

<sup>2</sup>The former meaning of trace is however extended in the trace algebra formalism we review here, as to also encompass untimed systems.

was first described in [31] and [32] introducing the concept of timed traces. 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 [33], [14] and [12]. In [14] and [12], in particular, a methodology has been developed for the automatic verification of speed-independent asynchronous circuits. This approach was further refined by Dill in [13] to include infinite traces for the representation of liveness properties ([5]).

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 [4]). Other models of concurrency can be found in the Mazurkiewicz traces [25] and partially ordered multisets (POM) [30]. All these results have also been extended to pure real-time systems (see [1] for an excellent survey on this topic).

## 3.2 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

### 3.2 The Trace Algebra Approach

---

of interest.

*Models of computation (MoCs)* have become in the recent years a standard way of representing different paradigms for computation, communication and also real-time issues [15]. 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 2, a first attempt to cope with such a problematic issue with the LSV model [23]. In this work several different MoCs were 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 lacked 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 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.

### 3.2.1 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  $y$ .

### 3.2 The Trace Algebra Approach

---

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<sup>3</sup> is that of **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 projection operator  $proj(B)(x)$  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 renaming operation is denoted by  $x' = rename(r)(x)$ , 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 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

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

---

<sup>3</sup>In the sense that it is the one which provides most information of systems behaviors

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

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

**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.

### 3.2.2 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.

### 3.2 The Trace Algebra Approach

---

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:

$$\Psi_l(P) = h(P) - h(\mathcal{C} - P)$$

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.

### 3.3 Comparison with other approaches

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.

### 3.3 Comparison with other approaches

---

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.

There is a strong similarity between the trace algebra approach to hybrid systems and the behavioral approach to dynamical systems description [29, 41]. A recent approach to modeling hybrid systems through the behav-

ioral approach can be found in [38] and [20].

The behavioral framework is strictly akin 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. For this reason, we devote the next section to describe the behavioral approach to system theory and the relationships with the trace algebra approach.

**Further Readings:** See [11], [10], [8] and [9].

## Chapter 4

# Behavioral Systems

### 4.1 The behavioral approach

The behavioral approach gives a formal framework for modeling and for the analysis of dynamical systems. Unlike the standard theory, the behavioral point of view does not make distinction 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.

Dynamical systems theory investigates the relations between different signals of time, i.e. *trajectories*; in this acceptance time is seen as an *independent* variable, while trajectory values are considered as *dependent* variables. A system's behavior is -roughly speaking- the set of all its possible trajectories.

Let us call  $T$  the set of independent variables <sup>1</sup> and  $W$  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

---

<sup>1</sup>In 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.

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. In this acception 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 distinction 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 differential 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:

## 4.1 The behavioral approach

---

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

where  $R_i$ , ( $i = 1, \dots, n$ ) 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:

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

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

### 4.1.1 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 equivalent *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.  $u^* \in \mathcal{C}^\infty(\mathbb{R}, \mathbb{R}^u)$ ), 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 in this acceptance considered as *free* variables, while output variables are *bound*, in the sense they can be completely determined by the inputs and their initial conditions.

### 4.1.2 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.

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  $(\mathbb{R}, W, \mathcal{B}(M))$ .<sup>2</sup>

---

<sup>2</sup>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.

## 4.1 The behavioral approach

---

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 codomain of the manifest variables is thus:

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

where  $E = \{e_1, \dots, e_{|E|}\}$  is the set of external terminals; the codomain of the latent variables, instead, is:

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

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

The full behavior  $\bar{\mathcal{B}}$  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**:

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

### 4.1.3 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  $dx/dt = f(x, u, t)$  we say that the system defined by the controlled vector field  $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \rightarrow \mathbb{R}^n$  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  $dx/dt = f(x, u, t)$  with initial condition  $x(0) = x_1$  yields  $x(t_F) = x_2$ .

In this acception, 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  $\bar{t} \in T, \bar{t} \geq 0$  and  $w \in \mathcal{B}$  such that  $w(t) = w_1(t)$  for  $t < 0$  and  $w(t) = w_2(t - \bar{t})$  for  $t \geq \bar{t}$ .

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).

## 4.2 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<sup>3</sup>.

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

---

<sup>3</sup>And indeed we will see in the next section that behavioral systems and trace algebras formalisms are strictly intertwined

## 4.2 Hybrid Systems and the behavioral approach

---

below. Nevertheless, we reckon this field of research is very promising, and expect that more general results will be available soon.

### 4.2.1 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*:

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

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:

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

$$(w_1, x_1) \wedge_t (w_2, x_2) \in \mathcal{B}$$

where  $\wedge_t$  is the concatenation operator at time  $t$ :

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

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:

$$\begin{aligned}x(t+1) &= f(x(t), u(t)) \\ y(t) &= h(x(t), u(t))\end{aligned}$$

for discrete-time systems, and:

$$\begin{aligned}\frac{dx}{dt} &= f(x(t), u(t)) \\ y(t) &= h(x(t), u(t))\end{aligned}$$

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

## 4.2 Hybrid Systems and the behavioral approach

---

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  $\mathbb{U} \times \mathbb{Y} \times \mathbb{X} \times \mathbb{X}$  for discrete-time systems and  $\mathbb{U} \times \mathbb{Y} \times \mathbb{X} \times T\mathbb{X}$  for continuous-time systems (where  $T\mathbb{X}$  is the tangent bundle of  $X$ ).

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  $\mathbb{U} \times \mathbb{Y} \times \mathbb{X} \times \mathbb{X}$ ; 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  $\mathbb{U} \times \mathbb{Y} \times \mathbb{X} \times T\mathbb{X}$ . 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.

### 4.2.2 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 [16],[17], while concatenability properties of behavioral hybrid systems can be found in [35].

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  $w_p \wedge_0 w_f \in \mathcal{B}$ .<sup>4</sup>

**Concatenability of behaviors.** The behavior  $\mathcal{B}_p$  is said to be *concatenable* with  $\mathcal{B}_f$  if, for any past trajectory  $w_p \in \mathcal{B}_p$ , there exists a future trajectory  $w_f \in \mathcal{B}_f$  such that  $w_p \wedge_0 w_f \in \mathcal{B}$ , where  $\mathcal{B}$  is the behavior resulting from the interconnection of behaviors  $\mathcal{B}_p$  and  $\mathcal{B}_f$  according to the scheme described in Section 4.1.2.

Conditions for checking the regularity of interconnections, i.e. the avoidance of impulsive phenomena are given in [35]. In the same paper it is also shown that if the "future" interconnection (represented by behaviors in  $\mathcal{B}_f$ ) models a regular feedback controller<sup>5</sup>, then the past interconnection behavior  $\mathcal{B}_p$  can be concatenated with  $\mathcal{B}_f$  for any choice of the constraints induced

---

<sup>4</sup>Recall the definition of the concatenation operator in the previous subsection.

<sup>5</sup>See [35] for a formal definition in the behavioral framework

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.

### 4.2.3 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 [26], 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 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:

$$\mathcal{B}_p \subseteq W^{\mathbb{N}_0}$$

where,  $W = W_{in} \times W_{out}$  and  $\mathbb{N}_0$  is the set of non negative integers.

The task of the supervisory controller is thus to restrict the plant behavior  $\mathcal{B}_p$  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  $\mathcal{B}_{spec}$ , called the **specification**, and, if  $\mathcal{B}_{cl} \subseteq \mathcal{B}_p$  is the closed loop behavior of the plant, to require that:

$$\mathcal{B}_{cl} \subseteq \mathcal{B}_{spec} \tag{4.1}$$

Clearly, if  $\mathcal{B}_{sup}$  is the supervisor behavior, then  $\mathcal{B}_{cl} = \mathcal{B}_{sup} \cap \mathcal{B}_p$ .

The control problem to be addressed in this framework is to find  $\mathcal{B}_{sup}$  such that the condition (4.1) is satisfied. We observe that such a problem is univoquely determined by the plant behavior  $\mathcal{B}_p$  and by the specification  $\mathcal{B}_{spec}$ , thus we refer to the pair  $(\mathcal{B}_p, \mathcal{B}_{spec})$  as a **supervisory control problem**. Moreover, a behavior  $\mathcal{B}_{sup}$  is called a **solution** to the supervisory control problem if

$$\mathcal{B}_{cl} = \mathcal{B}_{sup} \cap \mathcal{B}_p \neq \emptyset$$

and  $\mathcal{B}_{cl}$  satisfies the condition (4.1).

The control synthesis methodology for solving this problem is based on an abstraction process (see [26]): it takes the plant behavior  $\mathcal{B}_p$  and builds an abstracted behavior  $\mathcal{B}_{cal}$  such that  $\mathcal{B}_p \subseteq \mathcal{B}_{cal}$  (i.e. the abstraction includes all possible plant behaviors plus possibly some more). The abstraction process  $\mathcal{B}_{cal}$  is defined in order to make the synthesis problem easier, i.e. the supervisory control problem  $(\mathcal{B}_{cal}, \mathcal{B}_{spec})$  should be easier to solve than the starting problem  $(\mathcal{B}_p, \mathcal{B}_{spec})$ .

It can then be shown that the supervisory control  $\mathcal{B}_{sup}$  that solves the abstracted problem  $(\mathcal{B}_{cal}, \mathcal{B}_{spec})$  is also a solution of problem  $(\mathcal{B}_p, \mathcal{B}_{spec})$ .

### 4.3 Behavioral Systems and Trace Algebras

---

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  $\{\mathcal{B}_\theta\}_\theta$  be a collection of plants  $\mathcal{B}_\theta \subseteq W^{\mathbb{N}_0}$ , 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  $(\{\mathcal{B}_\theta\}_\theta, \mathcal{B}_{spec})$ , and its solution is the behavior  $\mathcal{B}_{sup}$  such that:

$$\mathcal{B}_{cl}(\theta) \subseteq \mathcal{B}_{spec} \quad (4.2)$$

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

It can be shown that, if  $\mathcal{B}_{sup}$  is a solution of the supervisory control problem  $(\cup_{\theta \in \Theta} \mathcal{B}_\theta, \mathcal{B}_{spec})$  then it is also a solution of the robust supervisory control problem and satisfies the condition (4.2).

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  $\theta$ .

### 4.3 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:

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

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
- $\mathcal{B} \subset W^T$  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:  $f : A \rightarrow \mathbf{A}^{\mathbb{R}}$  and, if  $v \in A$ , then  $f(v) : [0, \delta] \rightarrow \mathbb{R}$  (partial) or  $f(v) : \mathbb{R} \rightarrow \mathbb{R}$  (complete).

A *trace structure* is a pair:

$$(\gamma, P)$$

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

### 4.3 Behavioral Systems and Trace Algebras

---

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, in the acceptance of trace algebra formalism given in the previous chapter.

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

- a)  $\delta \in T$
- b)  $\mathbf{A} = W$
- c)  $A$  and  $\gamma$  are implicitly defined by  $\mathbf{A}$ . Indeed, if -for example-  $\mathbf{A} = \mathbb{R}^q$ , then:  $A = \{x_1, \dots, x_q\}$  and  $\gamma = (x_1, \dots, x_q)$ .
- d)  $f : A \rightarrow \mathbf{A}^{\mathbb{R}}$  is such that: if  $v \in A$ , then:  $f(v) \in \mathcal{B}$

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

Let  $g : g(Y) = (T, W, \mathcal{B})$  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 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  $\wedge_t$  (see Section 4.2.1).

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*.

**Further Readings:** [39], [40], [37], [41], [42], [35], [26].

## Chapter 5

# The Hybrid System Interchange Format

In this Chapter, describe the semantics of an interchange format proposed by Vanderbilt University as a tool to help designing hybrid systems. An interchange format is a file that contains data in a given syntax that is understood by different interacting tools. It is not a data base nor a data structure, but a simpler object whose goal is to foster the exchange of data among different tools and research groups. This interchange format has been used by the Chess group in Berkeley to build the HyVisual tool for graphical representation and simulation of hybrid systems.

The chances of success of this format are related to its semantics. If indeed its semantics are sufficient to capture and “protect” the different properties that come with the designs, then there will be no loss going from one framework to another due to the interchange format itself. The format is indeed a neutral go-between.

More ambitious is the Metropolis meta-model described in the next Chapter, since it is also used as mathematical model within the Metropolis framework. In this respect, the Metropolis meta-model is more powerful but it will require some degree of agreement with the design methodology that

it supports and with the general approach that contributed to its definition.

This Chapter is contributed in its entirety by the University of Pennsylvania MoBies group [27]. We thank Rajeev Alur and his co-workers to make it available to us.

## 5.1 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.

## 5.2 Syntax

### 5.2.1 Hybrid automaton

A hybrid automaton  $H$  is a tuple  $\langle S, s_0, V, P, T \rangle$ , 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, we use the term *location* for syntactic description and the term *state* for semantics definition.

**Variables and parameters.** We partition the set of variables of the automaton 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

## 5.2 Syntax

---

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

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, we have to ensure that the set of algebraic equations has 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  $x_i \rightarrow x$ , for  $i = 1 \dots n$ , whenever there is an algebraic equation  $x = f(x_1, \dots, x_n)$ , and then ensuring that the dependency graph is acyclic.

**Transitions.** A transition of a hybrid automaton is a tuple  $\langle s, g, \alpha, s' \rangle$ , 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 variables, executed atomically when the transition occurs. An assignment has the form  $x = f(x_1, \dots, x_n)$ , 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.** Before proceeding to describe a network of automata, we summarize the kinds of variables in an automaton and their

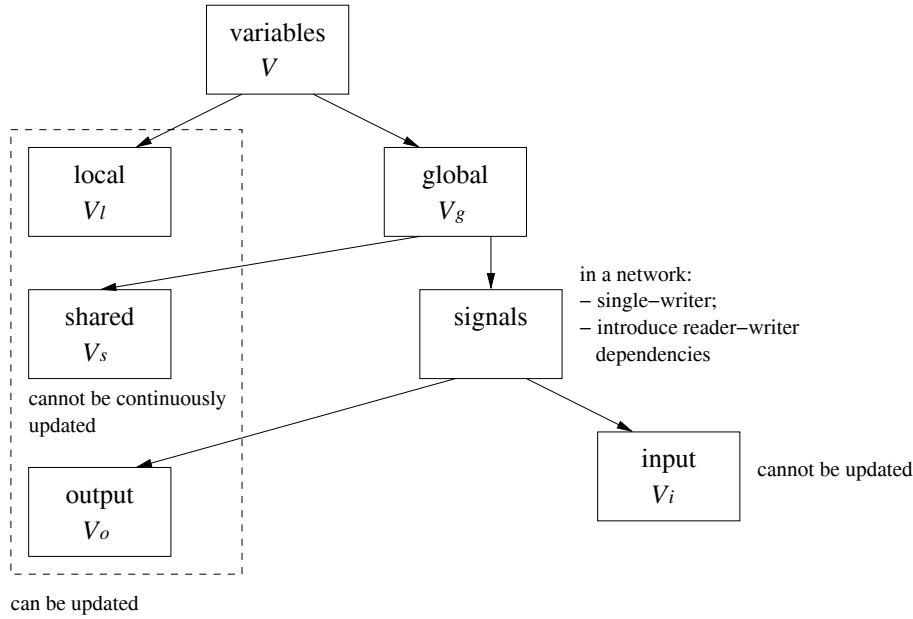


Figure 5.1: Variables and their uses

uses in Figure 5.1.

### 5.2.2 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  $V_o$ .

We impose the following consistency requirements:

- 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,  $V_i \subseteq \bigcup_{h \in HA} V_i^h$ .
- Local variables of the network are the union of local variables of the automata,  $\bigcup_{h \in HA} V_l^h = V_l$ , 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.** In order to obtain predictable executions of the network we construct a graph of dependencies between automata in the network. 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. We require that the graph of dependencies be acyclic.

## 5.3 Semantics

**Instantiated vs. parameterized automata.** Below, we give semantics to *instantiated* automata and networks of automata. Given a network, we obtain an instantiated network 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.

### 5.3.1 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  $Q(x)$ .

### 5.3 Semantics

---

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:

$$\langle s_1, Q_1 \rangle \xrightarrow{f_1} \langle s_1, Q'_1 \rangle \xrightarrow{e_1} \langle s_1, Q''_1 \rangle \xrightarrow{\alpha_1} \langle s_2, Q_2 \rangle \xrightarrow{e_2} \langle s_2, Q'_2 \rangle \xrightarrow{f_2} \dots$$

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

- If  $s$  has a differential equation  $\dot{x} = h(x_1, \dots, x_n)$ , then the first derivative with respect to time of  $f_x(t)$  equals to  $h(f_{x_1}(t), \dots, f_{x_n}(t))$ .
- If  $s$  has an algebraic equation  $x = h(x_1, \dots, x_n)$ , then the value of  $f_x(t)$  equals to  $h(f_{x_1}(t), \dots, f_{x_n}(t))$ .
- The invariant of  $s$  is satisfied; i.e.,  $i(f_{x_1}(t), \dots, f_{x_n}(t))$  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,  $g(Q(x_1), \dots, Q(x_n))$  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  $Q'$ .

## 5.3 Semantics

---

**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.

### 5.3.2 Semantics of the network

**State of the network.** A state of the network consisting of automata  $h_1, \dots, h_n$  is a tuple  $\langle \langle s_1, \dots, s$

## The Hybrid System Interchange Format

---

- $Q = Q_0$  and  $Q' = Q_n$ ;
- The action can be decomposed into  $n$  simpler actions  $\alpha = \alpha_1 \circ \alpha_2 \circ \dots \alpha_n$ ; 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  $V^{h_i}$ .

## 5.4 Notes and discussion

---

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, considering them 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.

**Pictorial representation of network.** In the examples below, we represent network 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.** The approach to the semantics taken in this document is to ensure that 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. However, during each continuous step, all automata will observe the same value for a shared variable.

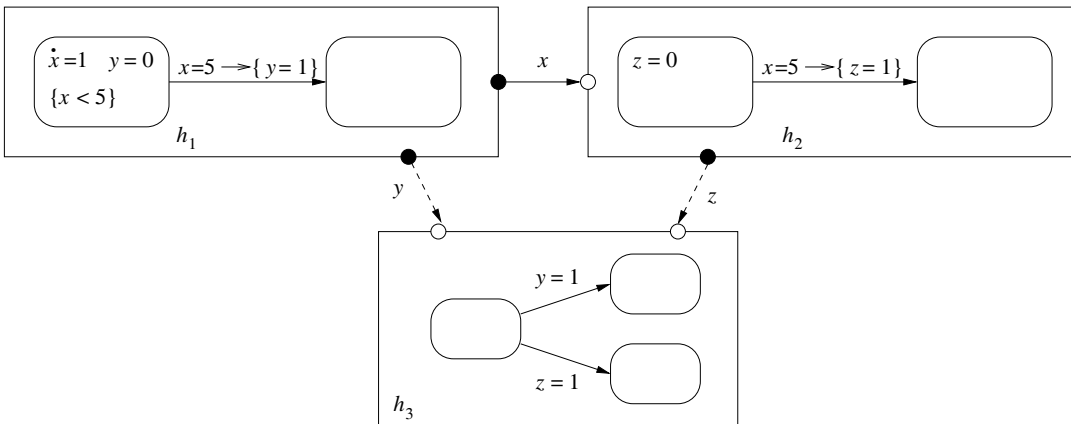
Dependencies between signals are captured to impose an execution order between automata. We require that each signal either is a global input to the network or is modified by exactly one automaton. Communication by means of signals between automata in a well-formed network does not introduce any nondeterminism 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 following network 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

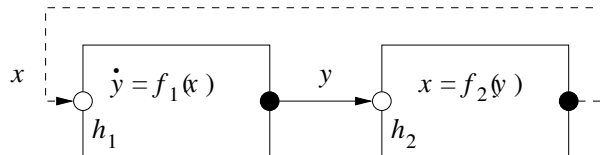
## 5.4 Notes and discussion

---

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.



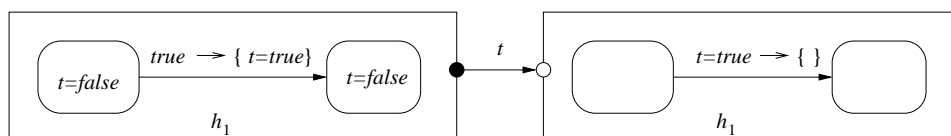
**Inputs used in differential equations are not dependencies.** The arguments of differential equations are not considered when constructing dependencies between automata. If an input is used only in a differential equation, say  $\dot{y} = f(x)$ , then  $y(t)$  does not depend on  $x(t)$  and thus does not constitute a dependency. The following example shows 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 our approach, we do not support events 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 shown below.



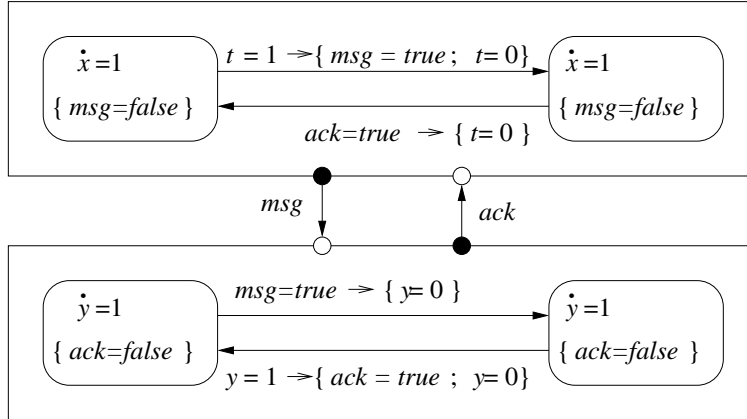
**Input constraints.** Consider the network of two single-state automata that represent two tanks that are being filled via the same pipe. The pipe is not captured by the model explicitly. The flows into the two tanks are the inputs to the network, but the sum of flows is bounded by the capacity of the 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

## 5.4 Notes and discussion

---

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 following network:



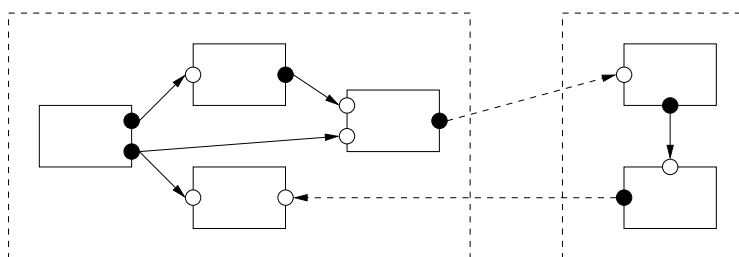
Notice that events  $msg$  and  $ack$  are always temporally separated. That is, at any time instance, only one of the two events can be updated. Still, the dependency graph, which is determined statically, has a cycle and this network is rejected by the current semantics. There are two alternatives 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 (similar to Statecharts) and some other synchronous languages, but it is a much more complex approach which we wanted to avoid.

**Purely asynchronous network.** Arguably, the system like this is more naturally modeled by asynchronous communication. If we replace signals  $msg$  and  $ack$  by shared variables, we obtain a purely asynchronous system.

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 a case where the graph of dependencies between automata consists of several disconnected subgraphs, which we call clusters. Within a cluster, automata communicate via signals, whereas communication between automata in different clusters is only by shared variables. Such a clustered network is schematically shown below.



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.

## 5.5 Conclusions

HSIF is a powerful interchange format. There are some semantic issues that may not appeal to designers and that may need attention. See [27] for a discussion on the semantics of HSIF. In particular, zero-time loops among FSMs are prevented to eliminate the risk of non deterministic behavior stemming out of a combination of deterministic subsystems. The

## 5.5 Conclusions

---

jury is still out to determine whether restrictions placed on the interchange format may hamper its adoption. We argue that the elimination of semantically unsound behaviors should be up to the tools and not to the interchange 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 6

# The Metropolis Framework

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 feature 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 workpackage of Columbus, but also implementation architectures thus allowing a designer to live his/her activity in only one

framework.

### 6.1 The Metropolis methodology: features and goals

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 [15]). Secondly, it uses a single formalism to represent both the embedded system and some abstract relevant characteristics of its environment and implementation platform. Finally, it separates orthogonal aspects, 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. 3.
- *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 independent of the functionality, by different groups (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.

Another fundamental aspect is the ability to specify rather than implement, execute reasonably detailed but still fairly abstract specifications, and finally use the best synthesis algorithms for a given application domain and implementation architecture. For these reasons in Metropolis it is represented explicitly the concurrency available at the specification level, in the

## 6.2 The Metropolis Meta-Model

---

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*.

## 6.2 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, 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 [23] 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 capture 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. We do 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, we provide mechanisms 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. In addition, the structure of the metamodel 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 [3]. 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.

### 6.2.1 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, communication, 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

## 6.2 The Metropolis Meta-Model

---

communication methods are defined so that they can be used to transfer information. The methods may evaluate and possibly alter the communication state. For the example of the 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 [28] 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 provides a mechanism that a quantity such as time or power can be defined and associated with actions, and constraints on the quantity can be specified in a form of predicate logic.

### 6.2.2 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 (i.e. terminated by `;`), 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 and 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 controls the events in its care set, but it is not affected by any other events. For a formal definition, see [36].

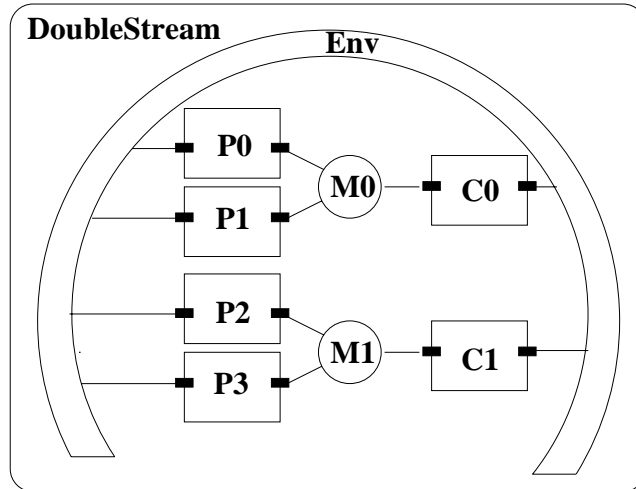


Figure 6.1: A netlist of processes and media. The object around the process is the environment, which is modeled by a medium in this example.

### 6.2.3 Processes and media

Fig. 6.1 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 specifications of the processes in Fig. 6.1 are given in Fig. 6.2-(a). 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

<pre> <b>process</b> IntX {   <b>port</b> IntReader port0;   <b>port</b> IntWriter port1;    IntX() { }    <b>void</b> thread() {     <b>int</b> x;     <b>while</b> (true) {   Rd:   x = port0.readInt();   Wr:   port1.writeInt(x);     } } } </pre> <p style="text-align: center;">(a)</p>	<pre> <b>interface</b> IntReader <b>extends</b> Port {   <b>update int</b> readInt();   <b>eval int</b> n(); }  <b>interface</b> IntWriter <b>extends</b> Port {   <b>update void</b> writeInt(<b>int</b> data);   <b>eval int</b> space(); } </pre> <p style="text-align: center;">(b)</p>
---	---

Figure 6.2: (a) Process used in Figure 1. (b) Interfaces used for the ports of the process (a).

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. 6.1 is shown in Fig. 6.2-(b). 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.

Fig. 6.3 shows the specification of the media used in Fig. 6.1. 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 of a function of the interface through the port will execute the code of the function provided in the medium.

```
medium IntM implements IntWriter,IntReader,IW,IR,IC,IS,IN {
  int storage, space, n;
  IntM() { 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 {}
```

Figure 6.3: Medium used in Fig. 6.1.

```
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 6.4: Process using an `await` statement.

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 metamodel 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 Fig. 6.4. 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 Fig. 6.4, an interface

## 6.2 The Metropolis Meta-Model

---

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. 6.4 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 section is executed. For example, in Fig. 6.4, 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.

### 6.2.4 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 Fig. 6.1 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

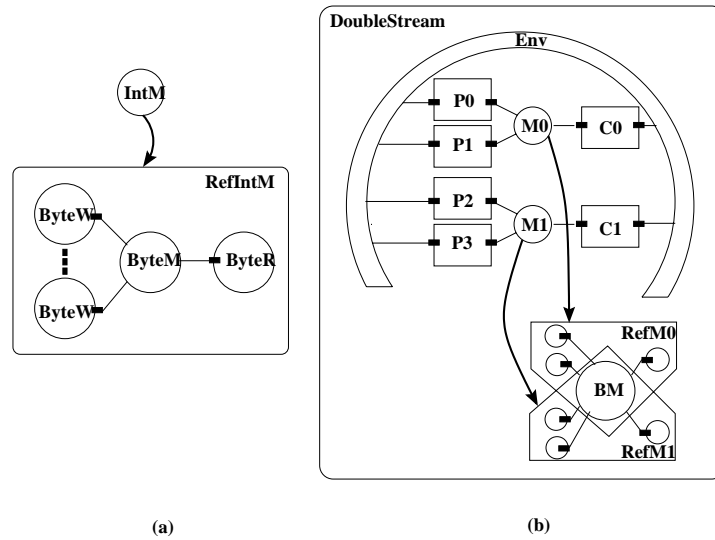


Figure 6.5: (a) A Refinement of the Medium `IntM`; (b) A Refinement of the netlist of Fig. 6.1 using the refinement netlist given in (a). The objects shown with names are instantiated in the netlist `DoubleStream`, while those without names are created inside the refinement netlists of (a).

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.

Fig. 6.5-(a) illustrates a refinement of the medium of Fig. 6.3. 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 system. The system designer, who first instantiates the original object to constitute his

## 6.2 The Metropolis Meta-Model

---

system behavior, then chooses a particular refinement netlist for the object to meet cost/performance goals.

In Fig. 6.5-(a), the refinement netlist `RefIntM` contains three types of media. `ByteM` is the medium with a byte storage. It implements interfaces called `ByteWriter` and `ByteReader`, which are identical with `IntWriter` and `IntReader` except that the size of data is byte. One object of this type is used in the refinement, which may be provided externally. `ByteW` implements the `IntWriter` interface so that each of its objects is connected from a port of an object (such as `P0` in Fig. 6.1) that originally accesses `IntM` with the interface.

The function `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 `writeByte()` of `ByteWriter` while ensuring the exclusion specified originally, i.e. no other process can execute the bodies of the functions of `IntWriter` implemented by the media in this refinement netlist during this period. `ByteR` is the third type, which implements the `IntReader` interface and is connected from a port of an object (such as `C0` in Fig. 6.1) that originally accesses `IntM` with the interface. As with `ByteW`, in the implementation of `readInt()`, the read function `readByte()` of `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 in Fig. 6.5-(b). Note that both refinement netlists (`RefM0` and `RefM1`) are instantiated with the same byte storage `BM` in place of `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 `ByteM` to be provided externally. While hierarchical refinements are excellent for modularity and re-use, they may be inefficient when it comes to implementation.

### 6.2.5 Coordination constraints

The specification given in Fig. 6.1 has two independent data streams. However, its refinement illustrated in Fig. 6.5-(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 Fig. 6.5-(b), it is possible to add the following code:

```
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.

## 6.2 The Metropolis Meta-Model

---

For instance,  $\text{end}(p, \text{BM.writeByte})$  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.

### 6.2.6 Quantity constraints

To specify performance and cost constraints in the meta-model, a quantity, such as time, power, or Quality of Service, must be defined first. This 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's 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 Fig. 6.2 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.

## 6.3 Conclusions

We presented 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 domain. 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 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.

**Further Readings:** See [3], [2], [36], [21], [22].

## Chapter 7

# Conclusions

In this report, we addressed the semantic of hybrid representations for providing a unified framework to analyze the present offerings. The ultimate aim is to propose the adoption of a particular modeling mechanism to be discussed in a third deliverable of the work package.

In our analysis, we analyzed the LSV tagged-signal model as a unifying denotational model and we noted its limitations. We then proposed the Trace Algebra approach that has more generality and theoretical power to analyze the composition of heterogeneous systems. We pointed out that this approach is strongly reminiscent of the Behavioral approach to System Theory even though more general. Finally we reviewed two modeling approaches that are strong candidates for adoption as interchange formats: HSIF and the Metropolis Meta-model.

The theoretical foundations of the meta-model rest partially on the trace algebra approach. We are presently strengthening the ties to provide a sound background to the ideas presented here. We believe that because of its generality, the Metropolis meta-model could provide the appropriate backbone for modeling hybrid systems. However, the continuous semantics has not been fully supported by the tool framework as yet and we have little if any experience in dealing with classical hybrid systems directly. We do

not see any fundamental issue here, however.

On the other hand, HSIF was created to support hybrid systems and as such it has a direct support for hybrid systems. We believe it will make sense to try to use them both at different phases and in different kinds of design. To do so, we will need more experience with modeling hybrid systems with Metropolis and to compare with more rigor the pros and cons of both approaches. In addition, we would like to include in the recommendation two other noteworthy European approaches: Modelica and Scilabs from INRIA. In both cases, much care has been given to the semantic aspects as opposed to some of the commonly used industrial tools and they are sufficiently general to include features that can make them strong candidates. They both are focused on the correct modeling for simulation while not much attention has been devoted to synthesis of implementations and formal analysis. We believe that the Metropolis project has more experience in this area. In all cases, the emphasis of HSIF on providing an *interchange* format may have an angle that the other approaches do not see as primary in their goals.

# Bibliography

- [1] R. Alur and T.A. Henzinger. Logics and models of real-time: A survey. In *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
- [2] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, and Y. Watanabe. Metropolis: an integrated design environment for electronic system design. *IEEE Micro*, 2003.
- [3] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogeneous systems. Technical Report 2002/01, Cadence Berkeley Laboratories, January 2002.
- [4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sept. 1991.
- [5] D.L. Black. On the existence of fair delay-insensitive arbiters: Trace theory and its limitations. *Distributed Computing*, 1(4):205–225, 1986.
- [6] S.D. Brookes. *A Model for Communicating Sequential Processes*. PhD thesis, Oxford University, 1983.
- [7] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating processes. In *NSF-SERC Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

- [8] J. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *Proceedings of the Second International Conference on Application of Concurrency to System Design*, Newcastle upon Tyne, UK, June 25-29 2001.
- [9] J. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Agent algebras. Working paper, 2002.
- [10] Jerry Burch, Roberto Passerone, and A. Sangiovanni-Vincentelli. Using multiple levels of abstractions in embedded software design. In T.A. Henzinger and C.M. Kirsch, editors, *EMSOFT 2001: Embedded Software, First International Workshop*, Lecture Notes in Computer Science 2211. Springer-Verlag, Tahoe City, CA, USA, 2001.
- [11] J.R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1992.
- [12] D.L. Dill. Trace theory for automatic hierarchical verification of speed independent circuits. In J. Allen and F.T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*. MIT Press, 1988.
- [13] D.L. Dill. Complete trace structures. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *Lectures Notes in Computer Science*. Springer-Verlag, 1989.
- [14] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

## BIBLIOGRAPHY

---

- [15] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.
- [16] A.H.W. Geers and J.M. Schumacher. Impulsive-smooth behavior in multimode systems, part i: State-space and polynomial representations. *Automatica*, 32(5):747–758, 1996.
- [17] A.H.W. Geers and J.M. Schumacher. Impulsive-smooth behavior in multimode systems, part ii: Minimality and equivalence. *Automatica*, 32(6):819–832, 1996.
- [18] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8), 1978.
- [19] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [20] A.A. Julius, S.N. Strubbe, and A.J. van der Schaft. Compositional modeling of hybrid systems with hybrid behavioral automata. Working Paper, 2002.
- [21] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: orthogonalization of concerns and platform-based design. Working Paper.
- [22] L. Lavagno, J. Moondanos, T. Meyerowitz, and Y. Watanabe. Modeling of architectural resources in metropolis. Internal document, Cadence, 2002.
- [23] E.A. Lee and A.L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 17(12):1217–1229, Dec. 1998.

- [24] J. Lygeros, C. Tomlin, and S. Sastry. Controllers for reachability specifications for hybrid systems. In *Automatica, Special Issue on Hybrid Systems*. 1999.
- [25] A. Mazurkiewicz. Basic notions of trace theory. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354. Springer-Verlag, 1989.
- [26] T. Moor, J.M. Davoren, and B.D.O. Anderson. Robust hybrid control from a behavioural perspective. In *Proc. 41st IEEE Conference on Decision and Control*, pages 1169–1174, Las Vegas, USA, December 2002. IEEE.
- [27] The University of Pennsylvania MoBIES Group. Hsif semantics(version 3, synchronous edition). Internal document, The University of Pennsylvania, August 22, 2002.
- [28] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual IEEE Symposium on Foundations of Computer Sciences*, pages 46–57, 1977.
- [29] J.W. Polderman and J.C. Willems. *Introduction to Mathematical System Theory: A Behavioral Approach*. Springer, New York, 1998.
- [30] V.R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, Feb. 1986.
- [31] G.M. Reed. A hierarchy of domains for real-time distributed computing. In *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.
- [32] G.M. Reed, A.W. Roscoe, and S.A. Schneider. Csp and timewise refinement. In J.M. Morris and R.C. Shaw, editors, *Fourth Refinement Workshop*. Springer-Verlag, Cambridge, England, 1991.

## BIBLIOGRAPHY

---

- [33] M. Rem, J.L.A. van de Snepscheut, and J.T. Udding. Trace theory and the definition of hierarchical components. In R.E. Bryant, editor, *Third Caltech Conference on VLSI*. Computer Science Press, Inc., 1983.
- [34] A.W. Roscoe. *A Mathematical Theory of Communicating Processes*. PhD thesis, Oxford University, 1982.
- [35] K. Takaba and J.C. Willems. Concatenability of behaviors in hybrid system interconnection. In *Proceedings of the 40th IEEE Conference on Decision and Control*, pages 370–375, 2001.
- [36] The Metropolis Project Team. *The Metropolis Meta-Model*, 0.3.0 edition, May 2002.
- [37] H.L. Trentelman and J.C. Willems. Control in a behavioral setting. In *Proceedings of the 35th IEEE Conference on Decision and Control*, Kobe, Japan, December 1996.
- [38] A.J. van der Schaft and J.M. Schumacher. Compositionality issues in discrete, continuous and hybrid systems. *Int. Journal of Robust and Nonlinear Control*, 11:417–434, 2001.
- [39] J.C. Willems. Models for dynamics. *Dynamics Reported*, 2:171–269, 1989.
- [40] J.C. Willems. Paradigm and puzzles in the theory of dynamical systems. *IEEE Transactions on Automatic Control*, 36:259–294, 1991.
- [41] J.C. Willems. On interconnections, control and feedback. *IEEE Trans. Automatic Control*, 42:326–339, 1997.
- [42] J.C. Willems. State construction in discrete event and continuous systems. In *Proceedings of the 41st IEEE Conference on Decision and Control*, Las Vegas, Nevada USA, December 2002.