UPMAIL Technical Report No. 153
February 1998
ISSN 1100–0686

Automata and Hybrid Systems

A seven-lecture mini-course delivered by

Professor Boris Trakhtenbrot

Department of Computer Science
Tel Aviv University

Visiting Researcher
Computing Science Department
Uppsala University

September—October 1997

Notes transcribed by

Faron Moller, Lars Thalmann, Gustaf Naeser, Paul Pettersson and Johan Bengtsson

Edited by Faron Moller and Boris Trakhtenbrot

Box 311, S-751 05 Uppsala, Sweden
Phone: +46-18-471 00 00
Fax: +46-18-51 19 25

Foreword

These notes represent a short (7 lecture) course on AUTOMATA AND HYBRID SYSTEMS presented at Uppsala University by Professor Boris Trakhtenbrot of Tel Aviv University. This course was presented during a two-month visit during the Autumn of 1997 to the Computing Science Department by Professor Trakhtenbrot, financed by a grant from The Swedish Foundation for International Cooperation in Research and Higher Education (STINT). This grant, titled LOGIC AND FORMAL METHODS IN COMPUTER SCIENCE, is aimed at promoting an exchange programme between Uppsala University and Tel Aviv University, exploiting the fact that the two communities are involved in various similar research areas, one of which being Timed and Hybrid Systems.

In the course, Professor Trakhtenbrot presented an approach to unifying the currently dynamic field of Hybrid Systems by extending and adapting the well-established theory of Automata. These notes were transcribed initially by students of the course, and later edited by Faron Moller and Boris Trakhtenbrot into a form suitable for quickly releasing as a Technical Report as a record of the ideas presented in the course. During the editing phase, the most visible inconsistencies were eliminated and the lectures were tied more tightly together; however, no attempt has been made at this stage to rework these notes into clearly defined textbook form. They are a record of the dynamic presentations, and need (and are worthy of!) later rewriting.

I would like to thank Professor Trakhtenbrot for presenting this course, and the students of the course for their participation, particularly those who agreed to the task of concentrating on the lectures to the extent of being able to provide the initial transcription.

Faron Moller
Computing Science Department
Uppsala University

February 1998

Boris Trakhtenbrot
Computing Science Department
Uppsala University
Sweden

Automata and Hybrid Systems
Lecture I: Monday 8 September 1997

Section 1

Introduction 2
Syllabus of the Course 4
Background Automata Theory 5
Retrospection, Residuals and Memory 7
Cardinalities and Topology 9
Labelled Transition Systems 11
About Systems and Nets of Equations 13
Logical Specifications 14
The Five Main Issues 15
Comments, History, Digressions 16

Introduction

Real-time and hybrid systems incorporate both discrete and continuous dynamics. The continuous aspects of these models may require incursions into areas of calculus (for example, into differential equations) which have little (if anything) in common with existing, well understood tools of automata theory and logic. However, these lectures deal only with issues which extend or adapt classical automata theory.

One way to extend the scope of automata theory is by interconnecting an automaton with an "oracle"; whatever the automaton can do while using this oracle is called its "relativization" with respect to this oracle. Another extension is with continuous time. These two extensions are conceptually orthogonal: the first one relies on the idea of interaction, which is visible already for discrete time automata; the second one focuses on the impact of continuous time without any assumptions about interaction. An appropriate combination of the two extensions facilitates a structured formalization of hybrid systems and a lucid adaptation (lifting) of the relevant heritage from classical automata theory.

Preliminary Requirements

Familiarity with basic automata theory, including some knowledge of automata on infinite objects and associated logics, is assumed.

However, in these lectures, there is no intention to enter into technical details of proofs. The concern is the explanation and motivation of the conceptual background and formulation of the basic facts.

As such, these lectures represent more of a lasting survey of the subject rather than a traditional (mini-)course. Thus there will be no commitment to concrete issues of, for example, verification.

The Challenge

Consider a hybrid system, such as that presented in Figure 1. In this system the Actuator sends continuous values to the Law (for example, temperature), and the Law responds by sending continuous values to the Sensor (for example, pressure computed as a function of the input temperature). While these values are continuous, the output of the sensor and the input to the actuator are both discrete values as these are the input and output to the Control which only handles discrete values.

To model this system we collectively refer to the Actuator, the Law and the Sensor as an *Oracle*. Disregarding what happens within the oracle, we consider only the (discrete-valued) input and output to the oracle (as observed by the finite-state control). This allows us to restrict our attention to discrete values.

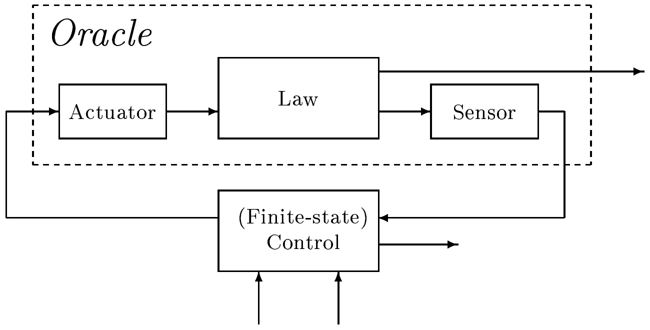


Figure 1: A hybrid System

Our goal with these lectures is to distill a lucid computational model. To accomplish this, we shall:

a) postpone heavy incursions into differential or integral equations as in control theory. In particular, we shall confine ourselves to discrete-valued oracles as proposed above.

b) avoid any premature mixture of semantics, syntax and pragmatics.

c) rely on "old-fashioned recipes" whenever possible, as advocated by Lamport. In particular, we shall approach definitional suggestions via incremental/orthogonal developments starting from finite automata (FA) and working our way towards a complete depiction of hybrid systems (HS) by adding fresh features, as depicted in Figure 2. Specifically, we

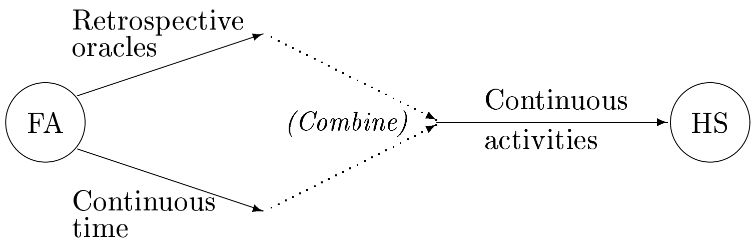


Figure 2: Creating the model

shall consider the two orthogonal conceptual avenues: oracles and continuous time.

d) adapt/lift the relevant heritage from classical automata theory.

So what then are the relevant fundamentals from automata theory intended for the further development of computational paradigms? We shall ultimately summarize these as "the five main issues" at the end of this lecture (see page 15.)

For explanatory and pedagogic reasons, these will be discussed with respect to three levels of semantics and syntax, which will be considered shortly (see page 5).

Syllabus of the Course

1. Survey of relevant fundamentals from automata theory.
    - Finite and infinite behaviour.
    - The convergence of three specification formalisms:
        (a) Labelled Transition Systems (LTSs);
        (b) Monadic Second-Order Logic (MSOL), for which popular temporal logics are a useful "syntactic sugar".
        (c) Circuits (Nets), interconnected networks of primitive elements.
    - Basic facts about determinization, uniformization and decidability.
2. Continuous-time transducers and acceptors.
    - Lifting automata theory from discrete to continuous time.
    - Illustrating phenomena not visible in the discrete case.
3. Overview and comparison of interaction paradigms.
    - Interacting automata (LTS).
    - Simultaneity versus interleaving.
    - Shared registers (variables) versus shared ports.
    - Restorability and feedback reliability.
4. Discrete transducers and acceptors with oracles.
    - Adapting basic facts from classical automata theory to automata with oracles.
    - The impact of choosing specific oracles.
5. Combining interaction with continuous time.
    - Analyzing real-time and hybrid systems around signal automata with finite memory and oracles for continuous processes.
    - A case study involving timed automata vs synchronous circuits over continuous time.
6. Badly timed elements and well timed nets.
    - A survey of an old (1964) unpublished report of R. McNaughton [13].
    - Discussion (suggested by the above-mentioned report) of "realistic" models for continuous time.

Topics 1, 3 and 4 are concerned with only the discrete time paradigm, while topics 2, 5 and 6 are concerned with the continuous time paradigm.

Topics 1, 3 and 6 represent the classical heritage, while topics 4, 2 and 5 represent this heritage lifted to the new computational paradigms explored in these lectures.

Background Automata Theory

We find it instructive to view the study of automata using the following three-level hierarchy.

Level 1: Finite behaviour

This level is referred to as the *micro approach* in [23]. Languages are sets of finite words, and there are two main formalisms.

- Labelled Transition Systems (LTSs) with accepting conditions;
- Regular expressions.

Level 2: Infinite behaviour

This level is referred to as the *macro approach* in [23]. ω-languages are sets of ω-sequences, and there are three main formalisms.

- LTSs with fairness conditions;
- ω-regular expressions;
- Temporal logics.

Level 3: Infinite behaviour with emphasis on operators (functions)

Here we consider ω-languages and ω-operators (that is, functions from ω-sequences to ω-sequences). The main formalisms here are as follows.

- LTSs as above;
- Systems of equations (Nets);
- Monadic Second-Order Logic (MSOL).

Regular expressions are not considered here as they are oriented specifically towards languages. Note also that MSOL was developed earlier than temporal logics.

Figure 3 provides a comparison of the micro and macro levels. The bottom of this figure illustrates how the equivalence of the three acceptor formalisms follows from the corresponding (algorithmic) translators (see [23]).

In the sequel, we will use the following notation. If u is a (finite) word, and w is an ω-word, then we denote by uw the *concatenation* of u and w, and we denote by u^ω the *infinite iteration* of u, that is the string $uuu\cdots$.

Before handling level 3 compare the following two different styles.

1. *From syntax to semantics:* First introduce a (syntactic) specification formalism (LTS); then use it to define appropriate semantic objects and characterize their properties.
2. *From semantics to syntax:* Provide direct definitions and characterizations (in set theoretic terms) of the intended objects without *a priori* commitment to specific formalisms. The choice of formalism is made later. (For details of this see [21].)

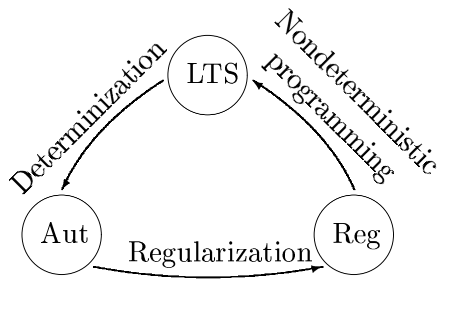
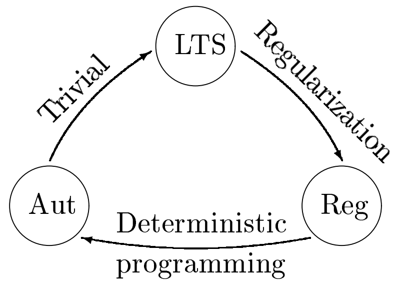
Micro	**Macro**
strings, words $\in X^*$ languages $\subseteq X^*$	ω-strings, ω-words $\in X^\omega$ ω-languages $\subseteq X^*$
Labelled Transition Systems (LTSs) (Nondeterministic Automata) *Micro anchoring*: initial state and set Q' of final states	**Labelled Transition Systems (LTSs)** (The same but with fairness conditions) *Macro anchoring*: initial state and set F of fairness conditions
Automata Automaton (deterministic LTS) with or without outputs	**Automata** The same but with fairness conditions
Regular expressions Regular operations and expressions	**ω-regular expressions** ω-regular operations and expressions
Translators  LTS Aut Reg Determinization Nondeterministic programming Regularization	**Translators**  LTS Aut Reg Trivial Regularization Deterministic programming
Decidability of Emptiness	Decidability of Emptiness

Figure 3: Automata theory: comparing the micro and macro approaches

When dealing with levels 1 and 2, style 1 is generally followed. However, we shall abide by style 2 throughout these lectures.

We continue with a precise definition of a Labelled Transition System and give a preliminary presentation of Nets and Logical Specifications. (The latter two will be formally defined later in a unifying approach covering both discrete and continuous time.)

Retrospection, Residuals and Memory

In the following we let either x_i or $x(i)$ denote the i^{th} element of a sequence $x \in X^\omega$.

Retrospection

An ω-operator $F : X^\omega \to Y^\omega$ is called *retrospective* iff, given $y = F(x)$, y_t depends only on $x_0x_1x_2 \cdots x_t$. F is *strongly retrospective* iff y_t depends only on $x_0x_1x_2 \cdots x_{t-1}$.

In terms of game theory, if we consider x to be the moves of the first player in a two-player game, and $y = F(x)$ to be the moves of the second player, then a retrospective operator F defines a strategy for the second player.

Residuals and Memory of Languages

Suppose that $P \subseteq X^\omega$ is an ω-language, and $u \in X^*$ is a finite string. Then the *residual* of P with respect to u, denoted P_u, is defined by:

$$P_u \quad = \quad \Big\{ w \ : \ uw \in P \Big\}$$

The *memory of* P is defined to be the number of different residuals of P, that is, the cardinality of the set $\{P_u \ : \ u \in X^*\}$.

Clearly, the memory of P is always at most countable. One case which will be of particular interest is when the memory is finite.

Residual and Memory of Operators

Suppose that $F : X^\omega \to Y^\omega$ is an ω-operator, and $u \in X^*$ is a finite string. Then the *residual* of F with respect to u, denoted F_u is defined by:

$$\begin{aligned} F_u(x) \quad &= \quad y_k y_{k+1} y_{k+2} \cdots \\ & \qquad \text{where} \quad y = F(ux) \\ & \qquad \text{and} \quad u = u_0u_1u_2 \cdots u_{k-1} \text{ is of length } k. \end{aligned}$$

The *memory of* F is defined to be the number of different residuals of F, that is, the cardinality of the set $\{F_u \ : \ u \in X^*\}$.

Examples of Retrospective Operators

Pointwise extensions

Given any function $f : X \to Y$, its *pointwise extension* $\mathrm{PE}^f : X^\omega \to Y^\omega$ defined by

$$\mathrm{PE}^f(x) \;=\; f(x_0)f(x_1)f(x_2)\cdots$$

is a retrospective operator with memory 1.

PROOF: $\mathrm{PE}^f(x)_t = f(x_t)$ and thus depends only on x_t; hence PE^f is clearly retrospective.

Furthermore, for any $u \in X^*$ and $w \in X^\omega$, $\mathrm{PE}^f_u(w) = \mathrm{PE}^f(w)$; hence PE^f has only one residual, and thus has memory 1.

Unit delay

The *unit delay* operator $\textsc{Delay}^a : X^\omega \to Y^\omega$ defined by

$$\textsc{Delay}^a(x) \;=\; ax$$

is a strongly retrospective operator with memory equal to the cardinality of X.

PROOF: $\textsc{Delay}^a(x)_0 = a$, and for $t > 0$, $\textsc{Delay}^a(x)_t = x_{t-1}$; hence $\textsc{Delay}^a$ is strongly retrospective.

Furthermore, $\textsc{Delay}^a_\varepsilon(x) = ax$ and $\textsc{Delay}^a_{ub}(x) = bx$; hence $\textsc{Delay}^a$ has exactly one residual for each element of X.

Square clock

The *square clock* operator $\textsc{SqClock} : \{0,1\}^\omega \to \{0,1\}^\omega$ defined by

$$\textsc{SqClock}(x)_t \;=\; \begin{cases} 1, & \text{if } \; x_0x_1x_2\cdots x_{t-1} \in (0+1)^*10^{k^2-1} \text{ for some } k > 0; \\ 0, & \text{otherwise} \end{cases}$$

is strongly retrospective and has a countably-infinite memory $\aleph_0$. (Informally, this function outputs a 1 at time t if exactly a square number of steps have passed since the input last registered a 1.)

<u>PROOF</u>: Clearly by definition this operator is strongly retrospective.

Furthermore, if we define the sequence y by

$$y_t \quad = \quad \begin{cases} 1, & \text{if} \quad t = k^2 \text{ for some } k > 0; \\ 0, & \text{otherwise} \end{cases}$$

then $\textsc{SqClock}_{10^n}(0^\omega) = y_{n+1}y_{n+2}y_{n+3}\cdots$; as these suffices to y are all different, these residuals $\textsc{SqClock}_{10^n}$ must all be different. Thus $\textsc{SqClock}$ has memory $\aleph_0$.

Cardinalities and Topology

Assume that $X = \{0,1\}$ is a binary alphabet.

1. The set of all ω-languages over X with memory 1 has cardinality $\aleph = 2^{\aleph_0}$.

   <u>PROOF</u>: For any $x \in X^\omega$, define the *tail language* L^x by
   $$L^x \quad = \quad \left\{ \, y \in X^\omega \quad : \quad x = uz \quad \text{and} \quad y = vz \quad \text{for some} \quad u, v \in X^*, \; z \in X^\omega \, \right\}.$$
   That is, $y \in L^x$ iff some tail of x and some tail of y coincide as ω-strings.
   L^x has memory 1 ($L^x_u = L^x$ for all $u \in X^*$); and there are $\aleph$ distinct tail languages.

2. The set of all finite memory functions from X^ω to X^ω has cardinality (at least) $\aleph$.

   <u>PROOF</u>: For any $x \in X^\omega$ define the *tail operator* $F^x : X^\omega \to X^\omega$ as follows.
   $$F^x(y) = \begin{cases} 1^\omega & \text{if} \;\; y \in L^x; \\ 0^\omega & \text{if} \;\; y \notin L^x. \end{cases}$$
   As above, F^x has memory 1, and there are $\aleph$ distinct tail operators.
   Note however that F^x is not retrospective.

3. The set of all finite memory retrospective functions from X^ω to X^ω has cardinality $\aleph_0$.

   <u>PROOF</u>: By the Fact on page 12, these are exactly the operators computable by finite-state transducers. The set of these transducers is certainly countable.

What was the idea with these exercises? Why are they about cardinalities? The answer is that if the cardinality is too large then the property is not strong enough.

Metrics

A full discussion on this subject can be found in [21].

Suppose we have an alphabet X with $\kappa \geq 2$ symbols. Then X^ω is a metric space with distance function ρ defined by

$$\rho(x,y) \quad = \quad \frac{1}{\kappa^{\mu(x,y)}} \quad \text{with} \quad \mu(x,y) = \min\{\, t \; : \; x(t) \neq y(t) \,\}$$

Hence usual notions can be used, for example the notion of continuous functions and the notions of closed and open sets. In particular we can note that retrospection implies continuity:

$$F : X^\omega \to Y^\omega \text{ is retrospective} \quad \text{iff} \quad \rho(Fx, Fy) \leq \rho(x, y).$$

$$F : X^\omega \to Y^\omega \text{ is strongly retrospective} \quad \text{iff} \quad \rho(Fx, Fy) \leq \tfrac{1}{\kappa}\rho(x, y).$$

The latter fact means that F has the Lipschitz's property, that is, F is a contracting map. The way is thus paved for using fixed point techniques.

Corollary: The cardinality of the set of retrospective functions is $\aleph$. (Retrospective functions are a subset of the countable set of continuous functions.)

Definition(Projective Languages)

The class Π of projective ω-languages is the minimal class which contains all the $safe$ (i.e. topological closed) ω-languages, and is closed under $\cap$, $\cup$, complement and projection π defined as follows: given $P \subseteq (X \times Y)^\omega$,

$$\pi(P) \quad = \quad \Big\{ x \; : \; (x, y) \in P \text{ for some } y \in Y \Big\}.$$

Claim: The cardinality of Π is $\aleph$.

Summary

In Table 1 the cardinality arguments are summarized. In this table, we consider ω-languages

	ω-operators		ω-languages	
(i)	all of them	$2^\aleph$	all of them	$2^\aleph$
(ii)	retrospective	$\aleph$	projective	$\aleph$
(iii)	finite memory	$\aleph$	finite memory	$\aleph$
(iv)	retrospective & finite memory	$\aleph_0$	projective & finite memory	$\aleph_0$

Table 1: Cardinalities of sets of ω-languages and sets of ω-operat ors

and ω-operators over finite alphabets. We can make the following observations.

(i) There are too many of them.

(ii)-(iii) Fewer, and with some good properties. However, they are still uncountable.

(iv) Our favourite objects are ω-operators with finite memory and projective ω-languages with finite memory. Both constitute countable sets.

Labelled Transition Systems

Syntax

A Labelled Transition System (LTS) T is a triple $\langle Q, \Sigma, \longrightarrow\rangle$ consisting of a set Q of states, a finite alphabet Σ, and a transition relation $\longrightarrow$ which is a subset of $Q \times \Sigma \times Q$. We write $q \xrightarrow{a} q'$ if $\langle q, a, q'\rangle \in \longrightarrow$. If Q is finite we say that the LTS is finite. We say that T is *deterministic* if for every state $q \in Q$ and every label $a \in \Sigma$ there exists at most one state $q' \in Q$ such that $q \xrightarrow{a} q'$. We say that T is *complete* if for every $q \in Q$ and $a \in \Sigma$ there exists $q' \in Q$ such that $q \xrightarrow{a} q'$.

Sometimes the alphabet Σ of T will be the Cartesian product $\Sigma_1 \times \Sigma_2$ of two other alphabets; in such a case we will write $q \xrightarrow{a,b} q'$ for the transition from q to q' labelled by the pair $\langle a, b\rangle$.

Let T be an LTS over the alphabet $\Sigma_1 \times \Sigma_2$. We say that T is Σ_1-*deterministic* if for every $q \in Q$ and $a \in \Sigma_1$ there exists at most one $q' \in Q$ and $b \in \Sigma_2$ such that $q \xrightarrow{a,b} q'$. We say that T is Σ_1-*complete* if for every q and a there exists q' and b such that $q \xrightarrow{a,b} q'$.

Warning. See that Σ_1-determinism is stronger than Σ-determinism, whereas Σ_1-completeness is weaker than Σ-completeness.

An ω-**acceptor** $\mathcal{A}$ over Σ is a triple $\langle T, \textsc{Init}(\mathcal{A}), \textsc{Fair}(\mathcal{A})\rangle$, where $T = \langle Q, \Sigma, \longrightarrow\rangle$ is an LTS over alphabet Σ; $\textsc{Init}(\mathcal{A}) \subseteq Q$ (the *initial states* of $\mathcal{A}$); and $\textsc{Fair}(\mathcal{A}) \subseteq 2^Q$ (the *fairness conditions* of $\mathcal{A}$).

A **transducer** of type $\Sigma_1 \to \Sigma_2$ is a pair $\langle T, q_0\rangle$, where T is an LTS over the alphabet $\Sigma_1 \times \Sigma_2$, which is Σ_1-deterministic and Σ_1-complete. Therefore, there are execution functions $\mathit{next} : Q \times \Sigma_1 \to Q$ and $\mathit{out} : Q \times \Sigma_1 \to \Sigma_2$ such that $q \xrightarrow{a,b} q'$ iff $q' = \mathit{next}(q, a)$ and $b = \mathit{out}(q, a)$.

Semantics

Note that unlike the *denotational* semantics of Monadic Logic and of Nets (to be given later), the semantics of LTS is *operational*. It relies on the consecutive execution of transitions which together generate a run, i.e., a finite or infinite sequence of steps. The acceptor semantics retains only those runs which meet the fairness/acceptance conditions.

A *run* of $\mathcal{A}$ is an ω-sequence $q_0 a_0 q_1 a_1 q_2 a_2 \cdots$ such that $q_i \xrightarrow{a_i} q_{i+1}$ for all i. Such a run meets the initial conditions if $q_0 \in \textsc{Init}(\mathcal{A})$. A run meets the fairness condition if the set of states that occur in the run infinitely often is a member of $\textsc{Fair}(\mathcal{A})$.

An ω-string $a_0a_1a_2\cdots$ over Σ is *accepted* by $\mathcal{A}$ if there is a run $q_0a_0q_1a_1q_2a_2\cdots$ that meets the initial and fairness conditions of $\mathcal{A}$. The ω-language *defined* or *accepted* by $\mathcal{A}$ is the set of all ω-strings accepted by $\mathcal{A}$.

Operators defined by transducers

If $T(q_0)$ is a transducer of type $\Sigma_1 \to \Sigma_2$ then for every ω-sequence $a_0a_1a_2\cdots$ over Σ_1 there exists a unique ω-sequence $q_0q_1q_2\cdots$ of states and a unique ω-sequence $b_0b_1b_2\cdots$ over Σ_2 such that q_0 is the initial state of $T(q_0)$ and $q_i \xrightarrow{a_i,b_i} q_{i+1}$ for all i. Therefore, one can associate with $T(q_0)$ the ω-operator that maps $a_0a_1a_2\cdots$ to $b_0b_1b_2\cdots$ as above. This ω-operator F is said to be *computed* by $T(q_0)$. Further, if Ω is the set of all subsets of the states of T, then the ω-language defined by the ω-acceptor $\mathcal{A} = \langle T, q, \Omega\rangle$ is (isomorphic to) the graph of F.

Note also that the Σ_1-determinacy of T implies that F must be retrospective. Furthermore, if T is finite-state, then F must be finite-memory.

PROOF: Given a sequence $a_0a_1a_2\cdots a_t \in \Sigma_1^*$, there is a unique sequence $q_0q_1q_2\cdots q_{t+1}$ of states of T and a unique sequence $b_0b_1b_2\cdots b_t \in \Sigma_2^*$ such that $q_0 \xrightarrow{a_0,b_0} q_1 \xrightarrow{a_1,b_1} q_2 \xrightarrow{a_2,b_2} \cdots \xrightarrow{a_t,b_t} q_{t+1}$. In particular, b_t depends only on $a_0a_1a_2\cdots a_t$.

The residual $F_{a_0a_1a_2\cdots a_t}$ is uniquely defined by the state q_{t+1}. Hence, if T is finite-state, then F must be finite-memory.

Conversely, if $F : \Sigma_1^\omega \to \Sigma_2^\omega$ is retrospective, then we can define a transducer *Trans*(F) which computes it as follows:

1. the set of states of the underlying LTS is $\Sigma_1^*/\cong$, where $u \cong v$ iff $F_u = F_v$;
2. the initial state is $[\varepsilon]_\cong$, the congruence class containing the empty string;
3. given $u = a_0a_1a_2\cdots a_{t-1} \in \Sigma_1^*$, the LTS will have the transition $[u]_\cong \xrightarrow{a,b} [ua]_\cong$, where $b = F(uaw)_t$ for some (any) $w \in \Sigma_1^\omega$, which is uniquely determined (by ua) due to the retrospectivity of F.

Clearly, if F has finite-memory, then *Trans*(F) is finite-state.

Summarizing the above, we have the following operational characterization of retrospective operators:

Proposition: (Transducer computability) An ω-operator is computable by a (finite-state) transducer iff it is a (finite-memory) retrospective operator.

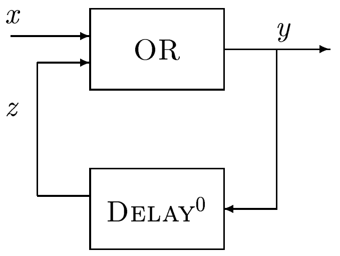
About Systems and Nets of Equations

As an alternative to the *operational* approach above, a retrospective operator can be characterized *denotationally* as the unique solution of an appropriate system of equations (*canonical equations*):

$$y(t) = \Phi(q(t), x(t)); \quad q(t+1) = \Psi(q(t), x(t)); \quad q(0) = \text{initial value} \qquad (*)$$

After some straightforward transformation, $(*)$ can be presented as a net (of equations).

A formal definition will be given later; for now we only give some intuition. For example, in Figure 4 we present both a net and its corresponding system of equations.



$$\begin{cases} y = \mathrm{OR}(x, z) \\ z = \textsc{Delay}^0(y) \end{cases}$$

Figure 4: A net and its system of equations

See that this system of equations defines uniquely y as a function g of x. Namely, for arbitrary $k = 0, 1, \ldots$, and arbitrary ω -sequence z:

$$g(0^k 1z) = 0^k 10^\omega$$

A set of functions $\mathcal{F}$ is a *basis* for a set of functions $\mathcal{G}$ if

1. $\mathcal{F} \subseteq \mathcal{G}$
2. For every $g \in \mathcal{G}$ there exists a system of equations Sys and $f_1, \ldots, f_n \in \mathcal{F}$ such that g is definable by Sys under the interpretation $f_1, \ldots, f_n$.

For example:

1. Conjunction and negation $\{\wedge, \neg\}$ form a basis for all boolean functions.
2. The pointwise extensions of conjunction and negation $\{\mathrm{PE}^\wedge, \mathrm{PE}^\neg\}$ to ω-strings over $\{0, 1\}$ together with the function $Delay^0$ form a basis for all finite memory retrospective operators over ω-strings.

Logical Specifications

Some historical milestones:

1. Propositional logic vs combinatorial circuits. Ehrenfest (1910), Shestakov (1935), Shannon (1938).
2. Beyond propositional logic: object quantifiers. McCulloch & Pitts and neural nets (1945), von Neumann and digital computing circuits (1943).
3. Toward Monadic Second-Order Logic. Church, Trakhtenbrot, Büchi, Elgot (c.1957). (A particular landmark is the Church lecture [6].)

The second order language $S1S$ is interpreted over the structure of the natural numbers with zero,successor function,and the usual ordering.It contains first order variables (ranged over by $t, \tau, ...$) for natural numbers,monadic second order variables (ranged over by $x, y, ...$). Typical atomic formulas are $t + 1 < \tau,\ x(t)$.

The formulas of $S1S$ are constructed from atomic formulas by logical connectives and first- and second-order quantifiers.

Temporal logics as syntactic sugar for S1S

We can define all of the usual temporal operators within our MSOL. For example, we have the following rough translations.

Temporal formula	**Desugared to**
$\Diamond p$	$\exists t.p(t)$
$\Box p$	$\forall t.p(t)$
$p\mathcal{U}q$	$\exists t[q(t) \wedge \forall t'.(t' < t \longrightarrow p(t'))]$

Thus for example, "weak fairness", $\Diamond\Box \mathit{enabled} \rightarrow \Box\Diamond \mathit{taken}$, would be rendered as

$$\exists t.\forall u.\Big(t < u \ \longrightarrow\ \mathit{enabled}(u)\Big) \quad\Longrightarrow\quad \forall t.\exists u.\Big(t < u\ \wedge\ \mathit{taken}(u)\Big)$$

In the above, we only use first-order quantification. However, consider the following example (due to Volper). We can express that x is a binary sequence with ones possible only at odd positions, $x \in \Big(0(0+1)\Big)^{\omega}$, as follows.

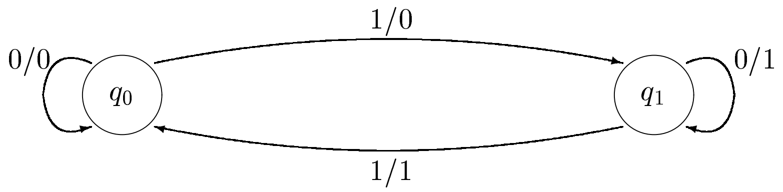
$$\exists u\{\neg u(0) \wedge \forall t(u(t) \leftrightarrow \neg u(t+1))\} \wedge \{\forall t(x(t) \rightarrow u(t))\}$$

This property cannot be specified without second-order quantifiers.

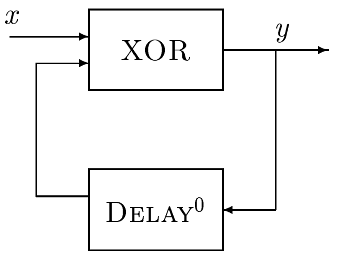
Illustrating three formalisms

1. Temporal logic
   Formula: $\exists^\infty t.\neg x(t) \rightarrow \exists^\infty t \neg y(t)$
   Temporal notation: $\Box\Diamond\neg x \rightarrow \Box\Diamond\neg y$

2. Automaton



3. Net (Circuit)



The Five Main Issues

We now introduce five issues from the theory of finite automata, which one would like to extend for other computational models. The first two issues presume appropriate boolean encoding of the involved alphabets and display the convergence of the three fundamental specification formalisms: automata, logic and nets. The fourth is based on the notion of uniformization.

Given a two-dimensional domain $D \subseteq X \times Y$, a total function $f : X \rightarrow Y$ *uniformizes* D iff the graph of f is a subset of the domain D (see Figure 5).

The five issues are then as follows

1. **Monadic logic** An ω-language is definable by a finite state ω-acceptor iff it is definable by a monadic formula (here – a formula in S1S).

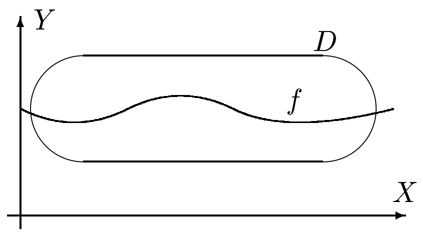


Figure 5: Uniformization

2. Nets The set $\{\mathrm{PE}^{\wedge}, \mathrm{PE}^{\neg}, \textsc{Delay}^{a}\}$ is a basis for ω-operators computable by finite state transducers.

3. Determinization An ω-language is definable by a finite state ω-acceptor iff it is definable by a deterministic finite state ω-acceptor.

4. Uniformization An ω-language L over alphabet $\Sigma_1 \times \Sigma_2$ which is definable by a finite state acceptor contains the graph of a retrospective operator iff L contains the graph of a retrospective operator which is computable by a finite state transducer.

5. Decidability of emptiness It is decidable whether the ω-language definable by a finite state acceptor is empty.

Comments, History, Digressions

Identification of the five issues in the Heritage:

1. Logic. Superissue.
2. Nets. Easy folklore.
3. Determinization. Intricate techniques for 1. Holds for Muller fairness conditions but not for Büchi fairness conditions.
4. Uniformization. Superissue.
5. Simple, but together with 1, immediately implies the fundamental fact: S1S is decidable.

History

Monadic Logic in Computer Science was identified and investigated as *the* temporal logic in 1957–1960 independently by Church, Trakhtenbrot and Büchi.

Trakhtenbrot gave a direct (set-theoretic) characterization of definability in S1S via retrospection, memory and topology (projective sets) and cardinality arguments. Partial answers for (i) and (iv) are given also in [21].

Full solutions of the superproblems 1 and 4 were given by Büchi.

Commenting on the Five Issues

Universality Arguments

$$\begin{array}{lccl} & \text{Monadic logic} & \Longleftrightarrow & \text{LTS-acceptors} \\ \text{to be completed by} & & & \\ & \text{Monadic logic} & \Longleftrightarrow & \text{ETL (extended temporal logic)} \end{array}$$

In the classical case, the convergence of the three formalisms above gives evidence to the *universality* of each of them. There are still further arguments in favour of monadic logic.

Decidability of Emptiness

Decidability of emptiness underlies the whole machinery of *model checking* and other types of algorithmic verification.

Uniformization

Uniformization underlies the golden dream of behavioural synthesis.

Given an implicit input/output specification $A(x, y)$, construct an explicit transducer Tr, which implements the specification, i.e., $\forall x \forall y\, [y = Tr(x) \Rightarrow A(x, y)]$

Two tasks: the existence of Tr; and the construction of Tr.

An analogy from calculus: given the differential equation $F(x, y, y')$ does there exist a solution $y = f(x)$, and how does it look like?

In our formalization: if there exists a retrospective solution there is also a finite-state solution.

Nets

Structural Synthesis. Given transducer T, construct net N over the given basis of primitives which implements T.

Behavioral and structural syntheses are perceived as two stages of comprehensive synthesis which leads from an implicit specification to the corresponding net (see [8, 23]).

Games and automata

For a more elaborate presentation consult [23], pages 109–115.

Consider an ω-sequence taken from $(X \times Y)^\omega \cong X^\omega \times Y^\omega$:

$$\langle x_0, y_0 \rangle \, \langle x_1, y_1 \rangle \, \cdots \, \langle x_t, y_t \rangle \, \cdots \quad \cong \quad \left\{ \begin{array}{l} x_0 \, x_1 \, \cdots \, x_t \, \cdots \\ y_0 \, y_1 \, \cdots \, y_t \, \cdots \end{array} \right.$$

This describes a two-player game history where x_i denote the moves of W (white) and y_i denote the moves of B (black).

Let $L \subseteq (X \times Y)^\omega$ be the subset of the plays in which B wins, as specified by a *Judge*.

A *strategy* for B is a retrospective operator $F : X^\omega \to Y^\omega$. Strategies are not assumed to have finite memory, but such strategies are preferred.

A *winning strategy* for B is one which assures that the result of the play, regardless of whites moves, will be accepted by the Judge. Such a strategy is an appropriate retrospective operator $F : X^\omega \to Y^\omega$ which uniformizes L.

In finite automata theory one considers finite state games, i.e., the Judge is a finite-state acceptor.

Claim: If B has a winning strategy, then it has also a finite state winning strategy.

This is a corollary from the Fundamental Theorem (ultimately the result of Büchi-Landweber, based on McNaughton's game theoretic approach), the solution of the Church synthesis problem.

Theorem:

1. In any finite-state game one of the players has a winning finite-state strategy.
2. There is an algorithm which given a finite-state game (i.e., a finite-state acceptor, the Judge) will:
   (a) determine which player has a winning strategy;
   (b) construct a winning finite-state strategy (i.e., a finite-state transducer) for this player.

Since there still are some difficulties when lifting with respect to interaction and/or continuous time, only the corollary, and not the whole theorem, has for now been included into the five main issues.

Boris Trakhtenbrot
Automata and Hybrid Systems
Section 2: Monday 15 September 1997

Computing Science Department
Uppsala University
Sweden

Section 2: Continuous Time

Signals 20
Languages and Operators 21
Examples of Non-Zeno Languages and Non-Zeno Operators 23
Relationships Between Properties of Operators 27
Variability and Latency 29

This lecture is an introduction to continuous time with signals $R^{\geq 0} \to X$ instead of ω-sequences $\in X^{\omega}$. The intention is to present the basic semantical universes; later (Section 3) appropriate specification formalisms will be developed for them. Retrospection and Memory are given as in the discrete case. The new property *Speed Independence* is presented with some basic facts and examples. The property of operators (functions) over languages (sets) is emphasized.

Signals

A function from $\mathbb{N}$ to Σ is called an ω-string over Σ. A function from $\mathbb{R}^{\geq 0}$ to Σ is called a *signal* over Σ, or a Σ-signal. A function ξ from the right open (!) interval $[0, d)$ into Σ is called a Σ-fragment of length d. The fragment ξ is simple iff it is constant on the open interval $(0, d)$. An important subset of the signals is the set of non-Zeno signals.

A signal x is *non-Zeno* (or *of finite variability*) if there exists an unbounded increasing sequence $t_0 = 0 < t_1 < t_2 \ldots < t_n < \ldots$ such that x is constant on every interval (t_i, t_{i+1}). In this case we say that the sequence t_i induces a scanning of x with (the corresponding sequence of) simple fragments.

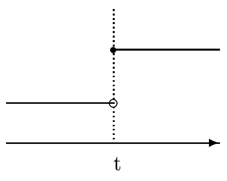


Figure 6: A Right-Continuous Signal

In Figure 7 we have the graph of a signal with discrete values over continuous time. In the figure

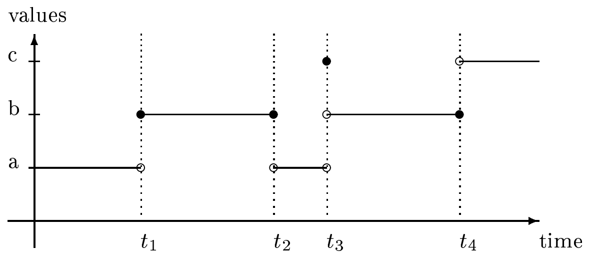


Figure 7: Non-Zeno signal over alphabet $\{a, b, c\}$

there are singularities (discontinuities) at the timepoints t_1, t_2, t_3 and t_4. The signal makes a "jump" at these points. During the time between two singularities the signal has a fixed value. We will refer to these time intervals with a fixed value as "steps". The signal itself alternates jumps and steps.

For any non-Zeno signal x and any $t > 0$ there is $u < t$ such that x is constant in the interval (u, t); we denote the value of x in this interval by $\textsc{LeftLim}(x)(t)$ or by $x(t-0)$. Similarly, for any t one defines $RightLim(x)(t)$ or $x(t+0)$ We say that x is *continuous* or *constant* at t if x is constant in some open interval (t_1, t_2) containing t: $t_1 < t < t_2$. Note that by definition no signal is continuous at 0. We say that x *changes* at t if x is not continuous at t.

Non-Zeno signals are more physically realistic than arbitrary signals. For example, the signal that has the value 0 at all irrational time moments and the value 1 at the rational time moments is not non-Zeno. The signal *ONLY5* which receives the value 0 at moment 5 and otherwise has the value 1 is non-Zeno.

More restricted classes of signals are characterized by restrictions imposed on the allowed discontinuities:

(i) **Right Continuity (RC)**. For every t there holds $x(t) = x(t+0)$ This is equivalent to the requirement: x allows a scanning with constant fragments.

(ii) **Left Continuity (LC)**. For every $t > 0$ there holds $x(t) = x(t-0)$

(iii) **Bursting (BS)**. Assume x is discontinuous at t.Then $x(t) \neq< x(t+0)$, and if $t > 0$ then also $x(t) \neq x(t-0)$.

Often in the literature *timed sequences* are used. A timed sequence is simply an ω-sequence

$$\begin{matrix} a_0 & a_1 & a_2 & \cdots & a_k & \cdots \\ t_0 & t_1 & t_2 & \cdots & t_k & \cdots \end{matrix}$$

over an alphabet X with increasing time stamps and $t_i \to \infty$. It can be described as a specific signal, namely, (as in Figure 8) the value of the signal at time t_i is a_i, but otherwise the value

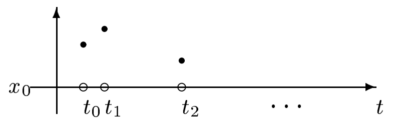


Figure 8: A Timed Sequence

is some "neutral" value $x_0 \in X$.

Languages and Operators

A set of ω-strings over Σ is called an ω language of type Σ. A set of signals over Σ is called a signal language of type Σ -

A function F from ω-strings over Σ_1 to ω-strings over Σ_2 is called an ω-operator of type $\Sigma_1 \to \Sigma_2$. A function F from signals over Σ_1 to signals over Σ_2 is called a *signal operator* of type $\Sigma_1 \to \Sigma_2$.

Above we considered properties (i.e. sets) of signals; these are first order properties. We are going now to consider also properties of languages and of operators, i.e. second order properties. But note, that every property of sets induces also a first order property (signals being represented by singleton languages), and a property of operators (represented by their graphs).

I. Properties with topological flavor.

1. Say that x is a limes signal of the language L if for each natural k there is a $x' \in L$ such that x and x' coincide on $[0,k]$. A signal-language L is *safe* (topological closed) iff it contains all its limes-signals.

2. F is said to be non-Zeno (Right-Continuous,Burst) if it maps non-Zeno (Right-Continuous,Burst) signals into non-Zeno (Right-Continuous. Burst) signals.

An operator F is retrospective (strongly retrospective) iff Fx and Fy coincide in the closed interval $[0,t]$ whenever x and y coincide in the closed interval $[0,t]$ (right open interval $[0,t)$.

Examples of non-Zeno retrospective operators

Pointwise extensions. Given any function $f : \Sigma_1 \to \Sigma_2$, the *pointwise extension* PE^f of f is defined as

$$\mathrm{PE}^f(x)(t) = f(x(t)).$$

Unit Delay The *unit delay* $\textsc{Delay}^a$ is defined as follows.

$$\textsc{Delay}^a(x)(t) \quad = \quad \begin{cases} a, & \text{if } t < 1; \\ x(t-1), & \text{if } t \geq 1. \end{cases}$$

II Stretchings.

Definition. A stretching of the Time Axis $R^{\geq 0}$ is an arbitrary 1-1 monotonic mapping : $R^{\geq 0} \to R^{\geq 0}$

(i) A signal ξ ρ-stretches (is the ρ -stretching of) signal x iff ρ is a stretching, and

$$\forall t(\xi(t) = x(\rho(t))).$$

(ii) A language L' ρ-stretches (is the ρ-stretching of) language L iff it consists exactly of all ρ-stretchings of signals in L. In particular, operator F' ρ-stretches (is the ρ-stretching of operator F iff its graph is the ρ-stretching of the graph of F.

(iii) A property (whether first or second order) is stretching invariant iff whenever an object has this property so do all the stretchings of this object.

(iv) A language (an operator) which coincides with every its stretching is said to be *speed independent*.

Informally, an operator F is speed independent if a stretching of an input signal causes an identical stretching of the output signal. For example, if $y(t) = x(2t)$ then $F(y)(t) = F(x)(2t)$.

Note that in discrete time the only bijection on $[0, \infty)$ is the identity function. Therefore every operator over ω-signals is trivially speed independent.

III Memory-oriented properties.

1. A signal-language L_u is the residual of a signal-language L with respect to a fragment u iff

$$x \in L_u \iff u.x \in L$$

2. A signal operator F_u is the residual of the signal-operator F with respect to a fragment u of length d iff

$$F_u(x)(t) = F(u.x)(t+d)$$

A language L induces an equivalence on fragments: $u \equiv_L v$ iff the languages L_u and L_v coincide. Similarly - for an operator F. The corresponding equivalence classes are called states of L (of F). Let Q be the set of states; its cardinality is called the memory of L (of F).

Examples of Non-Zeno Languages and Non-Zeno Operators

In this section we provide many examples of signal operators that illustrate the notions introduced above. Most of them will be instrumental in the forthcoming development of the subject.

It is easy to check that all properties we consider above are stretching invariant. Namely,

First order properties (properties of signals):

non-Zenoness i.e. Finite variability (FV), Right Continuity (RC), Left Continuity (LC), Bursting (BS).

Second order properties.

Properties of languages: Speed Independence, Safety, Memory.

Properties of operators. Speed Independence. Retrospection, Strong Retrospection, Memory.

Properties which are not stretching invariant are said to be metrical.

Recall that we can identify signals with singleton languages. The notions defined for languages are extended to signals through this correspondence. For example, we say that a signal has finite memory if it has only a finite number of distinct suffixes. Note also, that some (but not all of) the non-Zeno operators below preserve Right Continuity, and so their corresponding restrictions provide also examples of Right-Continuous operators.

1. Constant signals. Clearly they are speed independent and have finite memory.
2. Signals TICK and PERIODIC-TICK.

$$\textsc{Tick}(t) = \begin{cases} \textit{true}, & \text{if } t < 1; \\ \textit{false}, & \text{if } t \geq 1. \end{cases}$$

$$\textsc{Periodic-Tick}(t) = \begin{cases} \textit{true} & \text{if } 2k \leq t < 2k+1 \text{ where } k = 0,1,2\ldots; \\ \textit{false} & \text{otherwise.} \end{cases}$$

   These signals are not speed independent and have uncountable memory.
3. The existential quantifier (notation $\exists$) maps boolean signals to boolean signals as follows:

$$\exists(x)(t) = \begin{cases} \textit{true} & \text{if there exists } \tau \text{ such that } x(\tau) = True \\ \textit{false} & \text{otherwise} \end{cases}$$

   $\exists$ is not retrospective, however is is speed-independent and has finite memory.
4. The following retrospective operator is speed-independent and has countable memory

$$\textsc{Prime}(x)(t) = \begin{cases} \textit{true} & \text{if } x \text{ changes a prime number of times in interval } [0,t); \\ \textit{false} & \text{otherwise.} \end{cases}$$

Timers

First we give a background for converting timers which appear (implicitly) in the literature into our framework.

Clocks

Figure 9 shows a Clock, which has no input and a continuous-valued output.

(No input)
u
(Continuous values)
u
(time) t

Figure 9: A Clock

Actuators

Figure 10 shows an Actuator which is a clock reset by discrete burst signal input.

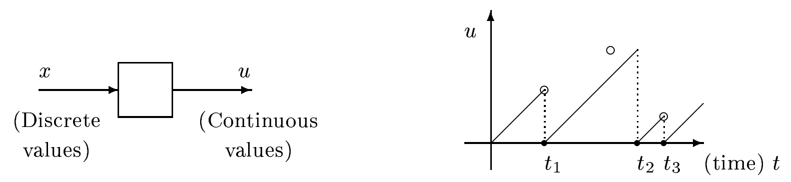
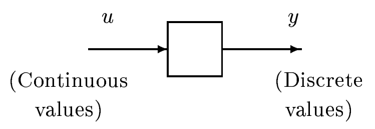


Figure 10: An Actuator



$$y(t) = \begin{cases} 1 & \text{if } u(t) \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Figure 11: A Sensor

Sensors

Figure 11 shows a Sensor, which converts a continuous-valued input into a discrete value by computing a threshold function.

Conventional Timers (As commonly presented in the literature)

Figure 12 shows a Timer. The input x and output y are discrete values, but the value u

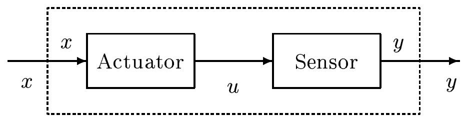


Figure 12: A Timer

within the Timer is continuous. Such Timers appear implicitly in the Timed Automata literature, see e.g. [4].

Timers (Our version of the above Timer)

TIMER (from [16]) maps signals to boolean signals as follows.

$$\text{TIMER}(x)(t) \quad = \quad \begin{cases} \textit{true} & \text{if} \quad t \geq 1 \text{ and } x \text{ was constant in the interval } (t-1, t] \\ \textit{false} & \text{otherwise} \end{cases}$$

The reset is triggered by jump changes (discontinuities) rather than by the occurrence of specific values. If x changes at all integer moments, y is constantly 0. Note that TIMER is retrospective (but not strongly retrospective!) and right continuous.

Periodic Timers

In [16] the signal operator PTIMER (Periodic Timer) was also considered; PTIMER maps signals to boolean signals and is defined as follows.

$$\textsc{PTimer}(x)(t) \;=\; \begin{cases} \textit{false} & \text{if } t < 1; \\ \neg\textsc{PTimer}(x)(t-0) & \text{if } x \text{ has changed value at } t-k \text{ and } x \text{ was} \\ & \text{constant in the interval } [t-k,t) \text{ for some} \\ & \text{integer value } k > 0. \end{cases}$$

Note, that PTimer is right continuous. The difference between PTimer and Timer is that PTimer is strongly retrospective, and if the input x has a finite set of singularities the output still has an infinite (periodic) set of singularities.

Filters

Filter (also from [16]) is defined as follows.

$$\textsc{Filter}^a(x)(t) \;=\; \begin{cases} a & \text{for } t < 1; \\ x(t-1) & \text{if } t \geq 1 \text{ and } x \text{ is constant in } [t-1,t); \\ \textsc{Filter}^a(x)(t-0) & \text{otherwise.} \end{cases}$$

The idea of a filter is that inertial delays, i.e. short steps of the input, are ignored. Filters are right continuous, strongly retrospective, and their output is without burst singularities even if the input is a burst signal.

Left Limit

The *left limit* operator LeftLim is defined as follows.

$$\textsc{LeftLim}^a(x)(t) \;=\; \begin{cases} a & \text{if } t = 0; \\ x(t-0) & \text{if } t > 0. \end{cases}$$

Last Jump Value

The *last jump value* operator LJV is defined as follows.

$$\text{LJV}^a(x)(t) \;=\; \begin{cases} a & \text{if } t = 0; \\ x(u) & \text{if } x \text{ is constant in } (u,t) \text{ and } x \text{ is not continuous at } u. \end{cases}$$

Last Jump Step

The *last jump step* operator LJS is defined as follows.

$$\text{LJS}^a(x)(t) \;=\; \begin{cases} a & \text{if } x \text{ is constant in } (0,t); \\ x(u-0) & \text{if there is } u \text{ such that } 0 < u < t \text{ and } x \text{ is constant} \\ & \text{in } (u,t) \text{ and } x \text{ is not continuous at } u. \end{cases}$$

Properties for the operators and languages which we have seen are summarised in Table 2.

Operator	Retrospective	Speed Independent	Cardinality of memory
Pointwise Extension PE	yes	yes	1
DELAY	strongly	no	$\aleph$
TICK	†	no	$\aleph$
PERIODIC-TICK	†	no	$\aleph$
$\exists$	no	yes	Finite
PRIME	strongly	yes	$\aleph_0$
TIMER	yes	no	$\aleph$
PTIMER	strongly	no	$\aleph$
FILTER	strongly	no	$\aleph$
LEFTLIM	strongly	yes	Finite
LJS	strongly	yes	Finite
LJV	strongly	yes	Finite

†TICK and PERIODIC-TICK are not operators, but languages consisting of one element each.

Table 2: Properties of operators and languages

Relationships Between Properties of Operators

First we remember that every retrospective ω-operator has at most countable memory. We observe that this fails for signal operators (e.g., for the DELAY operator), but we can easily check

Proposition: If a non-Zeno operator is retrospective and speed independent then its memory is at most countable.

The next fact is still about non-Zeno operators, but its proof is based on subtle technicalities concerning general signals:

Proposition: (A. Rabinovich : The Finite Memory Theorem) If a non-Zeno retrospective operator has finite memory, then it is speed independent.

Exercise: Reconsider the "Cardinality-Topology" exercises from page 9 for signal languages and signal operators (rather than for ω-languages and ω-operators). In particular, look to the case when the signal languages and signal operators are assumed to be (in addition) speed independent.

Exercise: Check that the class of retrospective signal operators is closed under superposition (i.e., composition). (The class of speed independent operators and the class of finite memory operators are also closed under superposition.)

We next capture the intuition of when a singularity at the output of a signal operator must be caused by a simultaneous singularity at the input.

Definition: Output Stability The signal operator F is *output stable* iff Fx is continuous at every moment t at which x is continuous.

Example: TIMER is not stable, since a change of the output can occur even if no changes at all occur in the input. The jump in the output may be caused merely by the elapse of time.

Exercise: Show that if F is speed independent, then it is also stable. (Note that the inverse is false.)

Exercise: Consider the equation + initialization:

$$q(t+0) \;=\; x(t+0) \;;\; q(0) \;=\; \textit{initial value} \tag{$*$}$$

See that for every non-Zeno signal x there are many non-Zeno signals q, which satisfy (*).But if q is required to be left continuous, then there exists an unique solution. Show, that this is not true when right continuity is required.

Residualizers

Let L be an arbitrary ω-language or singal languague of type Σ_1, and let Q be the set of its states.

Consider the operator R of type $\Sigma_1 \;\rightarrow\; Q$:

$R(x)(t) = \rho$ iff the residual of L wrt the fragment $x[0,t)$ is in the equivalence class ρ

R is said to be the residualizer of L. Similarly define the residualizer of an arbitrary operator F of type $\Sigma_1 \;\rightarrow\; \Sigma_2$. The properties below follow immediately from the definitions:

1) The residualizer R of L (of F) is a strong retrospective operator.

Remarks: (i) This observation points to the innate character of the notion "retrospective operator"; (ii) Residualizers are instrumental in proof techniques, omitted in the present exposition.

Properties preserved by the residualizer.

2) If F is speed independent so is R.

If F is a non-Zeno operator, R may happen not to be so. But,

3) If F is both non-Zeno and speed independent, the residualizer R is also speed independent and non-Zeno.

Variability and Latency

A signal x has *variability* $\leq k$ *("k-variability") in an interval* I if x changes at most k times in any subinterval $[t, t+1)$ of I. A signal has *variability* $\leq k$ if it has variability $\leq k$ in $[0, \infty]$.

Signals with k-variability seem to be more realistic than signals. Variability arguments are also relevant for decidability arguments for timed logics. So. let us comment more on variability and related notions.

A signal x has *fixed variability* in an interval I if it has k-variability in I for some k. A signal x has *latency* $\geq \delta$ *(δ-latency) in* I if x does not change at two points t_1 and t_2 ($t_1 < t_2$) in I with $t_2 - t_1 < \delta$. Note that if x has δ-latency then it has k-variability for each $k \geq l/\delta$.

Example: The boolean signal that has value *true* in $[k, k+1/k]$ ($k = 2, 3, \dots$) and *false* otherwise, has 2-variability; however there is no $\delta > 0$ for which the signal has δ-latency.

Example: The boolean signal x defined as

$$x(t) = \begin{cases} \textit{false} & \text{if } t < 1 \\ \textit{false} & \text{if there is an even } k \text{ such that } \sum_{i=1}^{k} \frac{1}{i} \leq t < \sum_{i=1}^{k+1} \frac{1}{i} \\ \textit{true} & \text{otherwise} \end{cases}$$

has unbounded variability.

Note that the output of the operators TIMER, PTIMER, FILTER have 1-variability, and TIMER and FILTER have 1-latency.

If $y = \text{DELAY}(x)$ and x has δ-latency in $[0, \infty)$ then y has δ-latency in $[1, \infty)$. Similarly, if x has k-variability in $[0, \infty)$ then y has k-variability in $[1, \infty)$.

Note that the output of a pointwise operator may have variability which is greater than the variability of all its inputs. Also note that even if boolean signals x_1 and x_2 have positive latency it might happen that for no positive δ is the positive disjunction of x_1 nd and x_2 of δ-latency.

An operator F has *fixed variability* if there exist a k such that for every x the signal $F(x)$ has k-variability.

F has *fixed latency* if there exists a $\delta > 0$ such that for every x the signal $F(x)$ has δ-latency.

Exercise: (easy) These two properties of operators are metrical (no stretching invariance!)

In Table 3 some properties are questioned about the four operators DELAY^a, TIMER, PTIMER

Operator	Strongly Retrospective	Fixed Variability	Fixed Latency
DELAY^a	YES	NO	NO
TIMER	NO	YES	NO
PTIMER	YES	YES	YES
FILTER	YES	YES	YES

Table 3: Properties of the operators DELAY^a, TIMER, PTIMER and FILTER.

and FILTER. Note that both PTIMER and FILTER have all the three desired properties.

The signal operators DELAY, TIMER, PTIMER and FILTER are not speed independent (they use metric properties of reals). Below we define the *Non-Metric Delay* operator NMDELAY which is speed independent and right continuous. First note that a signal over Σ is a step function from the reals to Σ. At every t, the operator NMDELAY outputs the value of the input at the preceding step. The formal definition of this operator is as follows.

$$\text{NMDELAY}(x)(t) = \begin{cases} a & \text{if } x \text{ was constant in } [0,t] \\ b & \text{if } x(t_1) = b \text{ and } x \text{ changed its value at } t_1 \text{ and } t_2 \text{ and} \\ & t_1 < t_2 \leq t \text{ and } x \text{ is constant in the intervals } [t_1,t_2) \text{ and } [t_2,t) \end{cases}$$

Note that NMDELAY is retrospective but not strongly retrospective. It is a stable, speed-independent and finite memory operator, but it is not a fixed variability operator.

Exercise: If a speed independent operator is *RC*, then it cannot be strongly retrospective. Hence, the operators PRIME, LEFTLIM, LJS, LJV are not RC-operators, because they are speed independent and strongly retrospective. See, that PTIMER, FILTER and DELAY are both *RC* and strongly retrospective operators.

Boris Trakhtenbrot
Automata and Hybrid Systems
Lecture II: Monday 15 September 1997

Computing Science Department
Uppsala University
Sweden

Section 3: Lifting the Trinity

Monadic Second-Order Theory of Order 32
Systems and Nets of Equations 34
Labelled Transition Systems (LTS) and Transducers 37
The Five Issues Revisited for Continuous Time 40
Comments About the Speed-independent Lifting 41

The objective of this lecture is to lift the five issues to continuous time. First we present more formally the "TRINITY" - three basic specification formalisms in Automata Theory. For each of the three an unique syntax is considered, but different interpretations make sense, depending on the kind of signals and operators in the intended semantical domains. The order of presentation in this lecture reflects the priority of denotational semantics over operational semantics. The semantics of Logical Formulas and that of Systems (Nets) of equations are given in denotational style. Both are equally persuasive for discrete and continuous time. The operational semantics of LTS in the discrete case extends straightforwardly to speed independent RC-signals but more care is needed with respect to non-Zeno signals. In the literature there are different modifications of LTS, aimed to do the job, and we comment on them.

Monadic Second-Order Theory of Order

Syntax

The language $L_2^<$ of monadic second-order theory of order has individual variables, monadic second-order variables, a binary predicate $<$, the usual propositional connectives, and first and second-order quantifiers. We use t, v for individual variables and x, y for second-order variables.

The atomic formulas of $L_2^<$ are formulas of the form: $t < v$ and $x(t)$. The formulas are constructed from atomic formulas by logical connectives and first and monadic second-order quantifiers.

We write $\Psi(x, y, t, v)$ to indicate that the free variables of a formula Ψ are among x, y, t, v.

Semantics

A structure $K = \langle \mathcal{A}, \mathcal{B}, <_K \rangle$ for $L_2^<$ consists of a set $\mathcal{A}$ (intuitively the set of time instances) partially ordered by $<_K$, and a set $\mathcal{B}$ of monadic functions from $\mathcal{A}$ into booleans $\mathbb{B}$ (intuitively signals).

We will be interested mainly in the following structures:

1. $\omega = \langle \mathbb{N}, 2^{\mathbb{N}}, <_{\mathbb{N}} \rangle$, where $2^{\mathbb{N}}$ is the set of all monadic functions from $\mathbb{N}$ into $\mathbb{B}$.
2. $\textsc{Real} = \langle \mathbb{R}, 2^{\mathbb{R}}, <_{\mathbb{R}} \rangle$, where $2^{\mathbb{R}}$ is the set of all monadic functions from $\mathbb{R}$ into $\mathbb{B}$.
3. $NZ\text{-}Sig = \langle \mathbb{R}^{\geq 0}, NZ, <_{\mathbb{R}} \rangle$, where NZ is the set of non-Zeno signals.
4. $RC\text{-}Sig = \langle \mathbb{R}^{\geq 0}, RC, <_{\mathbb{R}} \rangle$, where RC is the set of RC-signals.

We thus use the same syntax but different semantic interpretations.

The satisfiability relation

$$t_1, \ldots, t_m, x_1, \ldots, x_n \models_K \Psi\langle t_1, \ldots, t_m, x_1, \ldots, x_n \rangle$$

is defined in the standard way.

Note that in the structure ω the successor function and the order relation are mutually expressible by means of monadic second order logic.For example,, $t < t'$ is equivalent to:

$$\forall x \Big\{ x(t) \quad \wedge \quad [\forall t''(x(t'') \longrightarrow x(t''+1)] \rightarrow x(t') \Big\}.$$

The language l_2^{suc} interpreted in ω, widely known as $S1S$ (the monadic second order theory of one successor), is therefore equivalent to $L_2^<$.

Definability

Let $\Phi(x)$ be an $L_2^<$ formula and $K = \langle \mathcal{A}, \mathcal{B}, <_K \rangle$ be a structure. We say that a set $C \subseteq B$ is *definable* by $\Phi(x)$ iff $x \in C \Longleftrightarrow x \models_K \Phi(x)$; that is, $C = \{ x : x \models_K \Phi(x) \}$.

Examples

1. The formula

$$\forall t_1.\forall t_2.t_1 < t_2 \quad \wedge \quad \Big(\neg \exists t_3.t_1 < t_3 < t_2\Big) \longrightarrow \Big(x(t_1) \leftrightarrow \neg x(t_2)\Big)$$

defines the ω-language $\{(01)^\omega, (10)^\omega\}$ in the structure ω, the set of all NZ-signals in the structure NZ-*Sig*, and the set of all RC-signals in the structure RC-*Sig*.

2. The formula

$$\forall t_1.\forall t_2.t_1 < t_2 \quad \wedge \quad x(t_1) \quad \wedge \quad x(t_2) \longrightarrow \exists t_3.t_1 < t_3 < t_2 \quad \wedge \quad \neg x(t_3)$$

in the structure ω defines the set of strings in which between any two occurrences of 1 there is an occurrence of 0. In the signal structure NZ-*Sig* it defines the set of signals that receive the value 1 only at isolated points. Similarly in the structure RC-*Sig*.

3. The formula

$$\forall t_1 \forall t_2.t_1 < t_2 \quad \wedge \quad \Big(x(t_1) \leftrightarrow x(t_2)\Big) \longrightarrow \Big(\exists t_3.t_1 < t_3 < t_2 \quad \wedge \quad x(t_3) \leftrightarrow \neg x(t_2)\Big)$$

defines the empty language in the structures ω, NZ-*Sig* and RC-*Sig*. In the structure REAL it defines the set of functions that receive both the values 1 and 0 in every non-empty open interval.

In the above examples, all formulas have one free second-order variable x, and they define languages over alphabet $\{0, 1\}$. In general, a formula $\Psi(x_1, \ldots, x_n)$ with n free second-order variables defines a language over alphabet $\{0, 1\}^n$.

Exercise: Desugar $t = t'$.

Example: Consider the formula $A(x, y)$:

$$A(x, y) \overset{def}{\Leftrightarrow} [\exists t.x(t) \quad \wedge \quad \forall t.y(t)] \vee [\forall t.\neg x(t) \quad \wedge \quad \forall t.\neg y(t)]$$

This formula says that either x is true at some time and y is always true, or both x and y are always false. In each of the four structures, A defines the graph of an operator $y = F(x)$. Note that F is not retrospective.

In the following, we only consider the signal structures, *N-Sig* and *RC-Sig*.

Notation: $L(A)$ denotes the language accepted (defined) by the formula A.

Claim: For any formula A, the language $L(A)$ has finite memory and is speed independent.

To establish speed independence is easy, but to establish finite memory requires somewhat sophisticated reasoning.

We have only $\aleph_0$ formulas but $\aleph$ languages with finite memory and speed independence. Hence we need an alternative full characterization for languages which are $L_2^<$-definable. This can be done in different ways:

Denotationally (set theoretically): First, define projective languages as in the discrete case. Then one can prove that a language is $L_2^<$-definable iff it is projective, speed independent and has finite memory.

Operationally: This leads to the *LTS* machinery, i.e., automata theory.

Systems and Nets of Equations

Systems

Syntax. Let $y_1, \ldots, y_n$ (called *output variables*) and $x_1, \ldots, x_m$ (called *input variables*) be distinct variable names, and let F_i be $(n+m)$-ary functional symbols. Consider terms over those symbols, and a system *Syst* of equations which have the format $term1 = term2$.

Semantics. Let D be a set and let $f_i : D^{n+m} \to D$ be interpretations of the symbols F_i.

A tuple $\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle \in D^{n+m}$ *satisfies Sys* under the interpretation $f_1, \ldots, f_n$ iff $a_i = f_i(a_1, \ldots, a_n, b_1, \ldots, b_m)$ for all $i = 1, 2, \ldots, n$.

Sys defines the sequence of functions $g_1, \ldots, g_n : D^m \to D$ iff

$$\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle \text{ satisfies } Sys \iff a_i = g_i(b_1, \ldots, b_m) \text{ for all } i = 1, 2, \ldots, n.$$

In this case we say that *Sys* is (semantically) *well-defined* and that the g_is are *defined* by *Sys*.

It is clear that not every System *Sys* is well-defined. However, if *Sys* is well-defined, then the set of tuples that satisfy *Sys* is the graph of a function from D^m to D^n.

Remark 1: It was assumed above that all the functions F_i have the same arity. However, the extension to functions of distinct arity is straightforward. Also, instead of one domain D, many sorted functions may be considered.

Remark 2: In the case when the semantical domains consist of non-Zeno signals, it makes sense to use self explanatory syntactical sugar, which is based on explicit use of time-variables. Namely, use $x(t+0), x(t-0), \ldots$ for $RightLim(x)$, $LeftLim(x), \ldots$, and $G(x(t))$ for the corresponding Pointwise Extension $PE^G(x)$.

Example:

1) Alphabets: Q, Σ_1, Σ_2.

2) Functions: $\Phi : Q \times \Sigma_1 \Rightarrow \Sigma_2, \quad \Psi : Q \times \Sigma_1 \times \Sigma_1 \Rightarrow Q$.

3) System of *Canonical Equations* $CAN(\Phi, \Psi)$.

$$y(t) = \Phi(q(t), x(t)) \; ; \; q(t+0) = \Psi(q(t), x(t), x(t+0)) \; ; \; q(t) = q(t-0) \tag{1}$$

(Hint: Remember Canonical Equations in discrete time (Section 1).)

Appropriate desugaring to strong syntax would result in the system of equations below:

$$\begin{aligned} y &= PE^{\Phi}(q, x) \\ \pi &= RightLim(q) \\ \xi &= RightLim(x) \\ \pi &= PE^{\Psi}(q, x, \xi) \\ q &= LeftLim(q) \end{aligned} \tag{2}$$

Here x is declared to be the only input variable.

Nets

There are two additional requirements on the syntax of the equation system:

(i) Format of equations: the left hand side of an equation must be a single variable; the right hand side – a term with a single occurrence of a functional symbol.

(ii) No two equations may share the same left hand side variable.

Note that system (2) violates condition (ii), because π appears twice on the left side.

Variables occurring *only* on the right hand sides of an equation system are declared as *input* variables; all other variables are declared as *output* variables. Nets also use an alternative graphical syntax, as illustrated in Figures 13, 14. In Figure 14, y is an input variable, and x

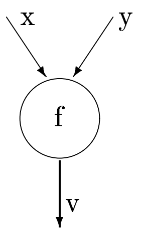
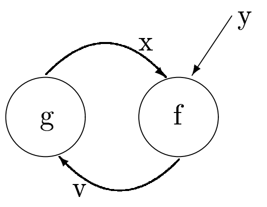


Figure 13: A pictured equation, $v = f(x, y)$



$$\begin{cases} v = f(x,y) \\ x = g(v) \end{cases}$$

Figure 14: An example net and its system of equations

and v are output variables.

To define the semantics, choose (arbitrary!) domains and a class $\mathcal{K}$ of functions which is *closed under superposition.*

Correspondingly, we will require that the solution should belong to the chosen class $\mathcal{K}$.

Example: Consider the net in Figure 14. Choose the domain of boolean values $\mathbb{B}$, and choose as class $\mathcal{K}$ all functions from boolean values to boolean values, i.e., all functions: $\mathbb{B}^m \to \mathbb{B}^n$. Interpret f as conjunction and g as negation. The system could then be expressed in the following way:

$$v = x \wedge y \qquad \text{and} \qquad x = \neg v$$

Then this net does not have any solution if the value 1 (*true*) is assigned to the input variable y; therefore it has no functional solutions. Thus this (interpreted) net is not well defined (not reliable).

Exercise: Prove that if a net contains no (feedback) loops, then it is reliable. (Remember that the class of functions $\mathcal{K}$ is closed under superposition.)

Exercise: Give examples of nets with many solutions.

The definitions above are presented in a general setting but we are mostly interested in the case when the class $\mathcal{K}$ consists of retrospective operators, whether in discrete or continuous time.

Canonical Equations for Retrospective Operators

Recall that in discrete time (see Section 1) the canonical equations appeared in the form:

$$y(t) = \Phi(q(t), x(t)); \quad q(t+1) = \Psi(q(t), x(t)); \quad q(0) = \text{initial value} \qquad (*)$$

and that every retrospective ω operator (and only such an operator) is definable by appropriate canonical equations:

The following continuous time version is less trivial.

Proposition. For every non-Zeno operator $F : \Sigma_1 \to \Sigma_2$, which is retrospective and speed independent, there exists a canonical system $CAN(\Phi, \Psi)$ (see equations (1) above) which defines F.

But remember that the canonical equations (even after desugaring) do not provide net-syntax, and, in addition, they use the nonretrospective operator *RightLim*.

A technical challenge is to get nets which are interpreted over retrospective operators and define the same operators as given canonical systems.

Labelled Transition Systems (LTS) and Transducers

Below, the operational semantics of LTS relies on the following kind of executions: a simple fragment ξ with duration d is applied in a state q_1 at time t, and forces the system to reach a state q_2 at time $t + d$. In this section (unlike the forthcoming section 6) only duration independent semantics is considered, i.e., q_2 depends only on the type of ξ. Hence, the duration information is syntactically (but in no means - semantically!) irrelevant, and the labeling reflects only the type of the fragment.We start with the simpler RC-semantics. In this case the syntax is, up to some mild restrictions,the same as in the discrete case.

1) RC-acceptors.

Let M be an acceptor (in the sense of Section 1), and $q_0 a_0 q_1 a_1 ...$ an accepted run. Assign to each a_i its duration d_i. Then the RC-signal

$$a = def\ a_0^{d_o}.a_1^{d_1}...$$

belongs by definition to the RC-language $L(M)$ accepted by M, iff $\sum d_i = \infty$.

The following claim is easy:

$L(M)$ is speed independent;

2) NZ-acceptors.

In the more general case of NZ-semantics, the type of a simple Σ-fragment is a pair (a_1, a_2) which we prefer to present (in order to improve readability) as a label $a_1 \bullet a_2$ from the alphabet Σ^2. Acceptors are handled again via concatenation of the corresponding simple fragments.

3) RC-transducers.

Let T be a transducer of type $\Sigma_1 \to \Sigma_2$ (in the sense of Section 1). We are looking now for an operational semantics, according to which T would compute a retrospective RC-operator F. Note that, in order to be consistent with acceptors, F should be speed-independent (as are the accepted RC-languages).

As a transducer (in the sense of Section 1), T is Σ_1-deterministic, Σ_1-complete, and, therefore, equipped with the execution functions *next* and *out*. However, because of duration nondeterminism, a given input signal x, may be scanned in different ways. Also duration-incompleteness, i.e., lack of appropriate non-Zeno runs, might a priori occur. The following condition is required just to face those problems:

$$next(q_1, a) = q_2 \quad \longrightarrow \quad next(q_2, a) = q_2 \ \wedge \ out(q_1, a) = out(q_2, a) \tag{3}$$

It is easy to see that, due to (3), T indeed defines a retrospective RC-operator F of type $\Sigma_1 \to \Sigma_2$. Moreover, the computation of F can be performed step by step applying the execution functions to an arbitrary RC-signal x. In this sense we say that T is an RC-transducer, which computes F.

4) NZ-transducers.

These transducers need more care. Again, according to the usual definition of a transducer T, there are functions *next* and *out* such that

$T(q_1, < a_1 \bullet a_2, b_1 \bullet b_2 >, q_2)$ holds iff

$$next(q_1, a_1 \bullet a_2) = q_2 \ \wedge \ out(q_1, a_1 \bullet a_2) = b_1 \bullet b_2 \tag{4}$$

But, as in the *RC* case, additional restrictions are needed. Namely,

(i) $$next(q_1, a_1 \bullet a_2) = q_2 \quad implies \quad next(q_2, a_2 \bullet a_2) = q_2$$

(ii) there exists a function $\Phi : Q \times \Sigma_1 \Rightarrow \Sigma_2$ such that in (4) one can replace

$$out(q_1, a_1 \bullet a_2) = b_1 \bullet b_2$$

by

$$b_1 = \Phi(q_1, a_1) \quad \wedge \quad b_2 = \Phi(q_2, a_2)$$

To summarize, a NZ-transducer T is fully characterized by two execution functions:

$$\begin{aligned} \Psi_T &: Q \times \Sigma_1 \times \Sigma_1 \to Q \\ \Phi_T &: Q \times \Sigma_1 \to \Sigma_2 \end{aligned} \tag{5}$$

such that

$$\Psi_T(q_1, a_1, a_2) = q_2 \quad \Rightarrow \quad \Psi_T(q_2, a_1, a_2) = q_2 \tag{6}$$

Namely, this is the pair for which:

$$\begin{gathered} T(q_1, a_1 \bullet a_2, b_1 \bullet b_2, q_2) \text{ holds} \\ \text{iff} \quad \Psi_T(q_1, a_1, a_2) = q_2 \quad \wedge \quad b_1 = \Phi_T(q_1, a_1) \quad \wedge \quad b_2 = \Phi_T(q_2, a_2) \end{gathered}$$

Operational vs Denotational Semantics.

Given a pair as in (5) – (6), associate it with two objects:

(i) the NZ-transducer T as above:

(ii) the canonical equations $CAN(\Phi, \Psi)$

Proposition. Both define the same speed-independent and retrospective NZ-operator.

Proposition (non-Zeno operators vs NZ-transducers). A non-Zeno operator F is computable by a NZ-transducer iff it is speed independent and retrospective; F is computable by a finite state NZ-transducer iff it is retrospective and has finite memory.

Proposition (*RC*-operators vs *RC*-transducers). Similarly, replace NZ everywhere by *RC*.

The Five Issues Revisited for Continuous Time

We consider first the NZ-structure.

1. **Monadic logic** An NZ-language L is definable by a finite-state NZ-acceptor iff it is definable by a formula in $L_2^<$ over the NZ-structure.
2. **Nets** The set $\{\textsc{LeftLim}, \mathrm{LJV}, \mathrm{LJS}, \mathrm{PE}\}$ is a basis for NZ-operators, computable by finite state NZ-transducers.
3. **Determinization** An NZ-language is definable by a finite-state NZ-acceptor iff it is definable by a deterministic finite-state NZ-acceptor.
4. **Uniformization** Let L be an NZ-language over alphabet $\Sigma_1 \times \Sigma_2$ which is definable by a finite-state NZ-acceptor. Then L is uniformizable by a retrospective NZ-operator iff it is uniformizable by a retrospective NZ-operator computable by a finite-state NZ-transducer.
5. **Decidability of emptiness** It is decidable whether the NZ-language defined by a finite-state NZ-acceptor is empty.

The five issues revisited for the RC-structure

We confine ourselves only to the second issue. For the others, replace NZ in the formulations above by RC.

2. **Nets** The set $\{NMDELAY, \mathrm{PE}\}$ is a basis for RC-operators which are computable in finite-state RC-transducers.

Discussion:

Below we describe acceptors based on Labelled Bitransition Systems, a formalism essentially suggested (in other terms) by Henzinger and his coauthors.

In a Labelled Bitransition System (LT_bS), both the edges and the nodes (states) are labelled.

Figure 15 depicts an example LT_bS, with states $Q = \{q_1, q_2, q_3\}$, alphabet $X = \{a, b, c, d, e\}$ and initial state q_1. Intuitively, we interpret edge labels as instantaneous steps, and node labels as lasting jumps.

A time scale is either:

1. an infinite unbounded sequence: $t_0 < t_1 < \cdots < t_n < \ldots$; or
2. a finite sequence: $t_0 < t_1 < \ldots < t_n$.

The associated sequence in which jumps alternate with steps is:

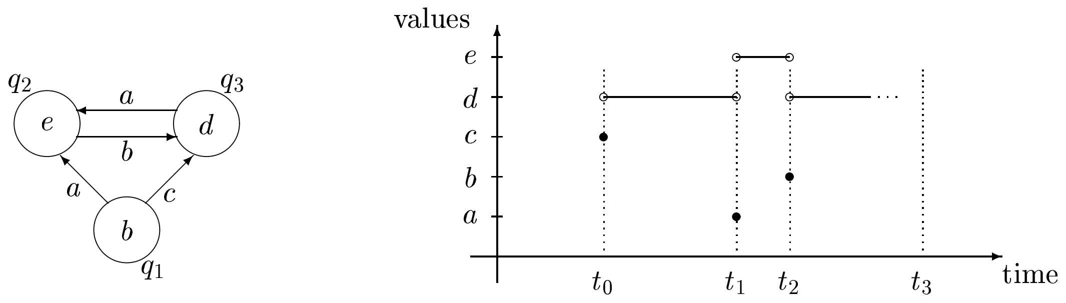


Figure 15: Example of a LT_bS and its supported non-Zeno signal X.

$$\begin{array}{lllllll}
q(t_0) = q_1 & q(t_0,t_1) = q_3 & q(t_1) = q_3 & q(t_1,t_2) = q_2 & q(t_2) = q_2 & q(t_2,t_3) = q_3 & \ldots \\
x(t_0) = c & x(t_0,t_1) = d & x(t_1) = a & x(t_1,t_2) = e & x(t_2) = b & x(t_2,t_3) = d & \ldots
\end{array}$$

This is a run which supports the non-Zeno signal X shown in Figure 15.

A transducer version can also be developed in this style.

Comments About the Speed-independent Lifting

1. Lifting essentially relies on the former results of Büchi for the two superissues wrt discrete time.
2. Requires non-trivial reconsideration of nets. Note that the RC-basis cannot be derived from the NZ-basis and requires specific treatment.

   See also [16] and [18].
3. The original version of the five issues also has some important algorithmic aspects, due to the algorithmic effectivity of the translations referred to.
4. The theory can also be adapted for burst signals and for timed sequences.
5. Uniformization has recently become a popular topic, and has relevance, for instance, in Control Theory.

Boris Trakhtenbrot
Automata and Hybrid Systems
Lecture IV: Monday 29 September 1997

Computing Science Department
Uppsala University
Sweden

Section 4: Interaction

Introductory Remarks 43
First Dichotomy of Interaction: Synchrony versus Asynchrony 44
Second Dichotomy of Interaction: Nets versus Webs 45
The Four Models of Manna and Pnueli 46
Petri Nets Revisited 49
Mutual Modelling of Asynchronous Nets and Webs 50
A Survey of Terminology 51
Restorability 53

In this lecture, we give an overview and comparison of some interaction paradigms for discrete time. In particular, we consider synchronous versus asynchronous communication paradigms, and shared variable versus shared ports (message-passing) paradigms. We include an analysis of the four models discussed by Manna and Pnueli [10].

Introductory Remarks

As interacting agents we consider (possibly infinite) labelled transition systems (LTS) with initial states. We do not assume any fairness restrictions. Finite LTS may be very large due to state explosion; hence specially-designed languages are used to provide a succinct and hopefully more comprehensible presentation. As an example, consider

$$q \quad \xrightarrow{x=1 \ or \ (y=z)} \quad q'$$

where x, y and z are booleans. This single edge represents 6 labelled edges from q to q'. More sophisticated programming languages can also be used. (For infinite LTS, that is the only way to have a finite representation.) Tools of this kind are important but they are *not* part of our model below, though we will use them when we find it convenient. We will even feel free to invent new ones.

Above we have already used a structured alphabet of actions $X \times Y \times Z$. Structured alphabets for states are also allowed. Let $M(q, a, a')$ denote a transition of an LTS, where $M(q, a, q')$ means that there is an edge in M labelled a which leads from state (node) q to state q'. This is pictured as:

$$q \xrightarrow{a} q'.$$

In the case of an LTS with structured alphabets we use notation such as $M(q_1q_2, a_1a_2a_3, q_1'q_2')$.

Structure is imposed on an LTS via a set of *registers* $Q_1, Q_2, \ldots, Q_n$, each one with its alphabet; and a set of *ports* $A_1, A_2, \ldots, A_k$, also with alphabets. We call them *register signatures* and *port signatures.* We also consider two kind of transitions, $\times$-transition and $||$-transitions. A $\times$-transition has the format

$$M(q_1q_2 \ldots q_m, a_1, \ldots, a_k, q_1'q_2' \ldots q_m') \qquad (\times)$$

whereas a $||$-transition has the format

$$M(q_1q_2 \ldots q_m, a_i, q_1'q_2' \ldots q_m') \qquad i \in 1, 2, \ldots, k \qquad (||)$$

From the two kinds of transitions we can construct two kinds of LTS.

1. $\times$-LTS that have transitions only of type $(\times)$ (synchrony, simultaneity, lockstep); and
2. $||$-LTS that have transitions only of type $(||)$ (asynchrony, interleaving, unlocked).

We consider anchored LTSs as ω-acceptors and ω-transducers. An *anchored* LTS is an LTS with so-called *anchoring requirements* on initialization and fairness conditions. For simplicity we will assume in this lecture LTSa with several initial states but with no restricting fairness conditions. For a given LTS L, let $L(M)$ denote the set of all runs which obey the anchoring requirements. Let $L_{act}(M)$ denote the corresponding set of ω-sequences of actions in $L(M)$, and let $L_{st}(M)$ denote the corresponding set of ω-sequences of states in $L(M)$. Classical automata theory usually considers L_{act}, however L_{st} can be considered as well.

Interaction is handled mostly with two combinators: composition and hiding. For simplicity we ignore for the moment the hiding combinator (a fierce troublemaker!).

We end this section with a discussion about the words: *reactive system*, *process*, and *distributed.* They are popular in the research community but still lack a precise underlying mathematical concept:

Milner On the word *process*: "... not mathematical objects whose nature is universally agreed upon. Not a well understood domain of entities" [14].

Abramsky On the words *reactive system*: "... a nice name for a natural class of computational systems. The search is still on for the best precise mathematical concept to explain these" [2].

Lamport & Lynch On the words *distributed computing*: "*Distributed* means spread out across space. Models of distributed computing are process models—concurrent execution of sequential processes; most notably via message passing" [9].

First Dichotomy of Interaction: Synchrony versus Asynchrony

In this section we compare synchronous versus asynchronous interaction between two LTSs.

$\times$-composition for $\times$-LTSs M_1 and M_2:

Given a $\times$-LTS M_1 with registers Q_1, Q_2 and ports A_1, A_2; and a $\times$-LTS M_2 with registers Q_2, Q_3 and ports A_2, A_3; construct $M \overset{def}{=} M_1 \times M_2$ with registers Q_1, Q_2, Q_3 and ports A_1, A_2, A_3 using

$$M(q_1q_2q_3, a_1a_2a_3, q_1'q_2'q_3') \overset{def}{=} M_1(q_1q_1, a_1a_2, q_1'q_2') \wedge M_2(q_2q_3, a_2a_3, q_2'q_4')$$

Note that A_1 and Q_1 are private for M_1; A_3 and Q_3 are private for M_2; and that A_2 and Q_2 are shared.

$||$-composition for $||$-LTS M_1 and M_2:

Given LTS M_1 and M_2 with the signatures defined above; construct $M \overset{def}{=} M_1||M_2$ with registers Q_1, Q_2, Q_3 and ports A_1, A_2, A_3 using

$$\begin{array}{llll}
M(q_1q_2q_3, a_1, q_1'q_2'q_3') & \overset{def}{=} & M_1(q_1q_2, a_1, q_1'q_2') \wedge (q_3' = q_3) & \text{if } a_1 \in A_1 \\
M(q_1q_2q_3, a_3, q_1'q_2'q_3') & \overset{def}{=} & M_2(q_2q_3, a_3, q_2'q_3') \wedge (q_1' = q_1) & \text{if } a_3 \in A_3 \\
M(q_1q_2q_3, a_2, q_1'q_2'q_3') & \overset{def}{=} & M_1(q_1q_2, a_2, q_1'q_2') \wedge M_2(q_2q_3, a_2, q_2'q_3') & \text{if } a_2 \in A_1 \cap A_3
\end{array}$$

Exercise: Show that these compositions are commutative and associative.

It follows from this exercise that these compositions may be considered for arbitrary sets $\{M_1, M_2, M_3, \dots\}$ of components.

Second Dichotomy of Interaction: Nets versus Webs

The second dichotomy reflects the restriction on the sharing of registers and ports between components:

Nets: components do not share registers, i.e., all registers of a component are private.

Webs: components do not share ports, i.e., all ports of a component are private.

Note that for nets, because the privacy of registers, it can be assumed (without loss of generality) that the components have unique registers. Similarly for ports in a web. Using the dichotomies of interaction and of restricted sharing, we get the following classification with four "architectures":

	$\times$	$\|\|$
Nets	1	2
Webs	3	4

Petri Graphs

A *Petri graph* is a bipartite graph with circle-nodes (places, registers) and box-nodes (transitions, ports). For an agent with one register (e.g. Q), or with one port (e.g. A) we will use pictures like the one shown in Figure 16.

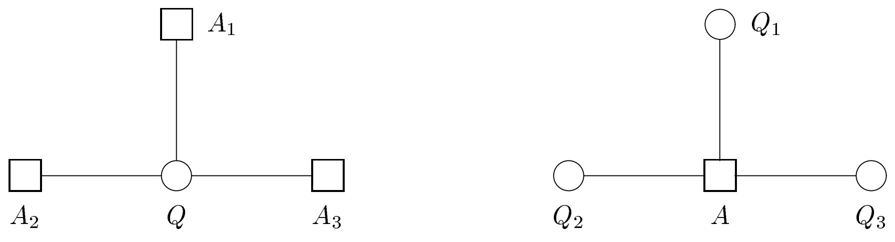


Figure 16: Nets versus Webs.

Example: Consider the Petri graph shown in Figure 17. It has two dual decompositions. Figure 18 shows the decomposition into a net over three agents which communicate via shared ports. Figure 19 shows the decomposition into a web over two agents which communicate via

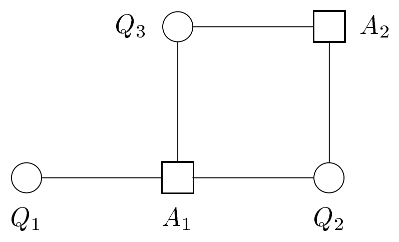


Figure 17: A Petri graph.

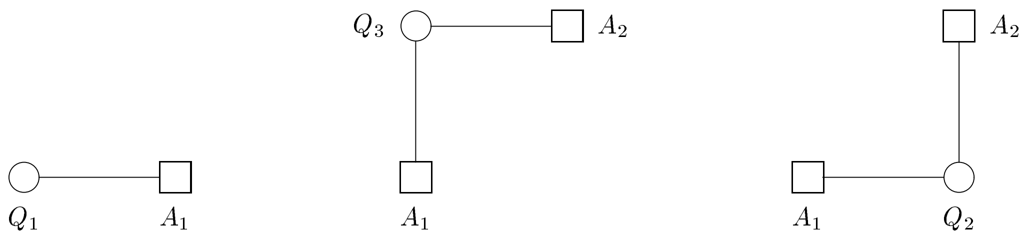


Figure 18: Decomposition into a net over three agents.

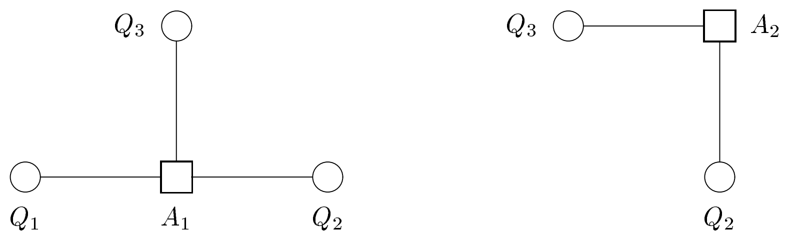


Figure 19: Decomposition into a web over two agents.

shared registers.

The Four Models of Manna and Pnueli

We consider here the four models introduced by Manna and Pnueli [10].

The Generic Model. We first consider the generic model in Section 1.1 (page 5) of [10]. It is (up to syntax) an LTS with structured alphabets. Nothing is said about how to compose agents. Rather, the effort is spent on describing the syntax. Meanwhile syntax is not a part of our model.

Model 1: Transition Diagrams. (Section 1.2, page 12). Now composition is introduced to the model, which corresponds to our $||$-webs (interleaving webs) up to syntactical details.

Model 2: Shared Variable Text. (Section 1.3, page 21). Another syntax for Model 1 (Transition Diagrams, above). It is still a $||$-web (up to syntax), so we do not distinguish this model from Model 1.

Model 3: Message Passing Text. (Section 1.10, page 70). This is (again, up to syntax) what we have called ||-nets.

Model 4: Petri Nets. (Section 1.11, page 86). In [10] this model is claimed to be "radically different" from the other three models because "it is not a programming language." Later, we will reconsider Petri nets as a model of interaction and see that it fits nice into our classification of models.

Note that ×-webs and ×-nets are not considered in [10].

An Example

Figure 20 is a reproduction of Figure 1.11 (program Binom) in [10]. It is not a Petri graph.

in	k, n	**: integer where** $0 \leq k \leq n$
local	y_1, y_2	**: integer where** $y_1 = n, y_2 = 1$
out	b	**: integer where** $b = 1$

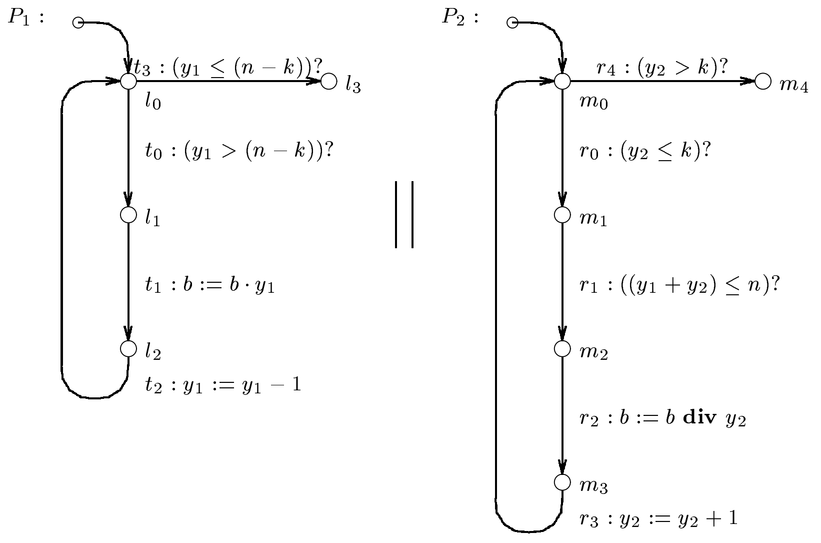


Figure 20: Program Binom - transition diagram.

Rather than illustrating the interface between the interacting components, it shows the diagram for each agent (LTS). The corresponding web for the Binom program is shown in Figure 21. The registers of P_1 are $\{n, k, \pi_1, y_1, b\}$ and the registers of P_2 are $\{n, k, \pi_2, y_1, y_2, b\}$. The value domain of π_1 is $\{l_0, l_1, l_2, l_3\}$ and the value domain of π_2 is $\{m_0, m_1, m_2, m_3, m_4\}$. It should be noticed that in [10] also the actions t_0, t_1, t_2 and t_3 for P_1, and r_0, r_1, r_2, r_3 and r_4 for P_2 are shown. However, since there are no shared actions for P_1 and P_2, pure shuffle is performed. Thus, we can forget about actions (or hide them) when we study state sequences only, i.e. when

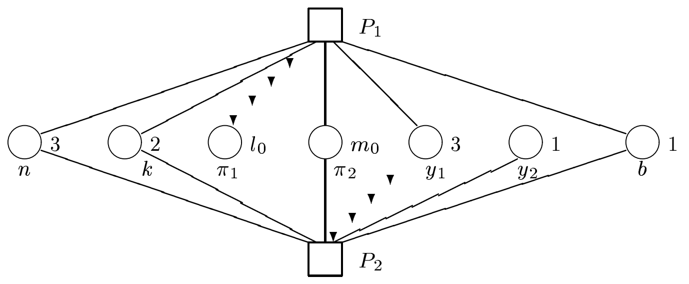


Figure 21: Program Binom - web for $P_1||P_2$.

we study how the common states of P_1 and P_2 develop. An alternative way to present the transition diagram in Figure 20 is to write each agent as tuples of its variables (connected with constrained transitions). The agent P_1 is (partly) shown in Figure 22.

$$\begin{tikzcd}
\bullet \arrow[d, bend left] & \\
(n,k,l_0,y_1,b) \arrow[r, "\text{if } y_1 \leq n-k"] \arrow[d, "\text{if } y_1 > n-k"] & (n,k,l_3,y_1,b) \\
(n,k,l_1,y_1,b) \arrow[d] & \\
(n,k,l_2,b\cdot y_1) \arrow[d, dash, dashed] & \\
{} &
\end{tikzcd}$$

Figure 22: Program Binom - P_1.

Petri Nets Revisited

Consult [10] for a description of Petri nets. In a Petri net, a transition is fired when the preplaces have at least one token each (Figure 23). When a transition fires, the numbers of tokens are

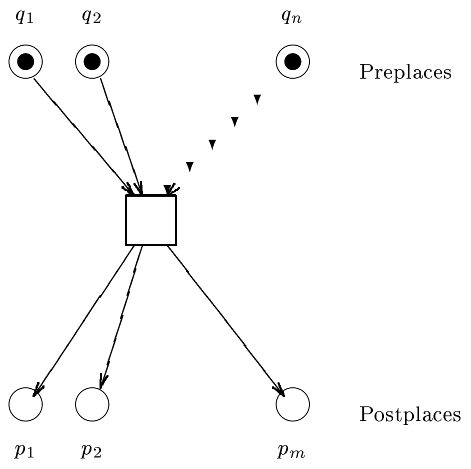


Figure 23: Illustration of fire rule for safe Petri nets.

decremented in all of the preplaces (i.e,, $q_1, q_2, \ldots, q_n$ in Figure 23) and incremented with one in all the postplaces (i.e. $p_1, p_2, \ldots, p_m$ in Figure 23). A particular case of Petri nets are the so-called *Safe Petri Nets* which may contain at most one token in each location. Therefore, each time a transition is fired in a safe Petri net, the postplaces must be empty.

Our goal is to analyse a Petri net as a system of interacting LTS-components (agents). There are (at least) two solutions: as a web, or as a net. (Recall how the graph in Figure 17 was decomposed in two different ways.)

Decomposing Petri Nets as Webs

Paradoxically, this was the first proposed way to decompose Petri nets. (The later decomposition of Petri nets as nets is due to Mazarkiewicz.) The corresponding LTS is shown in Figure 25. Each register has two possible values, empty or full, and each port has two possible values, active or passive. The transition

$$q_1 q_2 p_1 p_2 \quad \xrightarrow{active} \quad q_1' q_2' p_1' p_2'$$

is enabled iff $q_1 = q_2 = 1$; $p_1 = p_2 = 0$; $q_1' = q_2' = 0$; and $p_1' = p_2' = 1$. We then take the $||$-composition of such components.

Decomposing Petri Nets as Nets

A decomposition of a Petri net into nets is shown in Figure 25. The corresponding LTS has the transitions $q \xrightarrow{abcd} q'$ iff

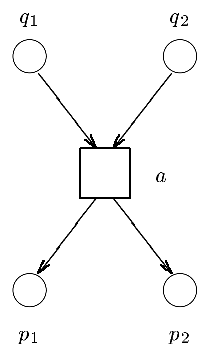


Figure 24: A component of a Petri Net which is decomposed as a Web.

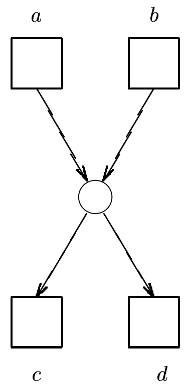


Figure 25: A component of a Petri Net which is decomposed as a Net.

- $q = 1 \ \wedge\ q' = 0 \ \wedge\ [a = b = 0] \ \wedge\ [c = 1 \Leftrightarrow d = 0]$; or
- $q = 0 \ \wedge\ q' = 1 \ \wedge\ [c = d = 1] \ \wedge\ [a = 1 \Leftrightarrow b = 0]$.

We then take the $||$-composition of such components.

Exercise: Check that both representations (decompositions) are faithful.

Mutual Modelling of Asynchronous Nets and Webs

Again look at the Petri graph of Figure 17 and assume it pictures a net N over three components. In [18] it was shown that dually it pictures also an appropriate web W which is equivalent to N in the sense that they accept the same ω-languages (i.e., the same set of ω-sequences of actions).

Exercise: Prove this.

The proof is quite easy. However, the reverse direction, i.e., to model webs as nets, is much harder and a more sophisticated approach is needed. The moral is that webs are more expressive in some sense than nets. Nevertheless, nets are preferred for other reasons, which will be explained later.

A Survey of Terminology

Table 4 surveys terminology and notation used in the literature. Each row in the table gives the names used by the different authors for the four architectures we have considered above.

Authors	$\times$ -nets	$\|$ -nets	$\times$ -webs	$\|$ -webs	Comments
Chandy & Misra	Synchronous Parallel 1) circuits 2) systolic arrays Synchronous Processes	Synchronous Distributed 1) messge passing 2) stream processing Asynchronuos Processes	Synchronous Shared variables (under ‘‘write’’ / ‘‘read’’ consistency)	Asynchronous Shared variables	1) ‘‘union’’ for pure $\|$. 2) Enjoying notational machinery of prog. languages for LTS. 3) State semantics for programs. Also Dataflow.
Manna & Pnueli		Message Passing Text		1) Transition Diagrams 2) Shared Variable Text	1) Generic model: basic transition systems. 2) Petri Nets out of classification.
Francez		Distributed Programs		Shared Variable Programs	1) "parallel" used as "generic". 2) Somtimes: concurrent $\Longleftrightarrow$ shared variabels.. 3) Interleaving "called" asynchronous. computation.
Apt & Olderog		Distribued Programs		Parallel Programs	1) "concurrent" used as "generic"
Milner	Synchrony	Asynchrony			

Table 4: A Survey of Terminology/Notation in the Literature.

Restorability

We have earlier classified interaction paradigms as

	$\times$	$\vert\vert$
nets	1	2
webs	3	4

Nets have disjoint (private) registers, and webs have disjoint (private) ports.

Notation: For a given agent M,

$L_{run}(M)$	denotes the set of all runs;
$L_{act}(M)$	denotes the set of all action traces;
$L_{st}(M)$	denotes the set of all state traces;
$L(M)$	denotes the generic form.

Restorability

Given $L(M_1)$ and $L(M_2)$ (but not the M_i themselves!), is $L(M_1 oper M_2)$ uniquely defined?

- For synchronous nets (1) L_{act} is restorable but L_{st} is not.
- For synchronous webs (3) L_{act} is not restorable but L_{st} is.
- For asynchronous webs (4) nothing is restorable (see the example on below).

A Precautionary Example

Assume we have a web $H||P$. Restorability is about the unique definability of the language $L(H||P)$ for the composed web from the languages $L(H)$ and $L(P)$ of its components (or similarly, restoring $L_{act}(H||P)$ from $L_{act}(H)$ and $L_{act}(P)$, or $L_{st}(H||P)$ from $L_{st}(H)$ and $L_{st}(P)$). Does it hold? The answer is "no".

Example: Three agents M_1, M_2 and N are represented in Figure 26 by their Petri graphs and by their LTS. They have $\textsc{Init}(M_1) = \textsc{Init}(M_2) = q_1$ and $\textsc{Init}(N) = q_2$. Observe that $L(M_1) = L(M_2)$, $L_{act}(M_1) = L_{act}(M_2)$ and $L_{st}(M_1) = L_{st}(M_2)$. $M_1||N$ and $M_2||N$ are webs, but $L(M_1||N) \neq L(M_2||N)$. For example,

$$\begin{aligned} q_1 a q_2 b (q_3 c)^\omega &\in L(M_1||N) \setminus L(M_2||N); \\ abc^\omega &\in L_{act}(M_1||N) \setminus L_{act}(M_2||N); \\ q_1 q_2 q_3^\omega &\in L_{st}(M_1||N) \setminus L_{st}(M_2||N). \end{aligned}$$

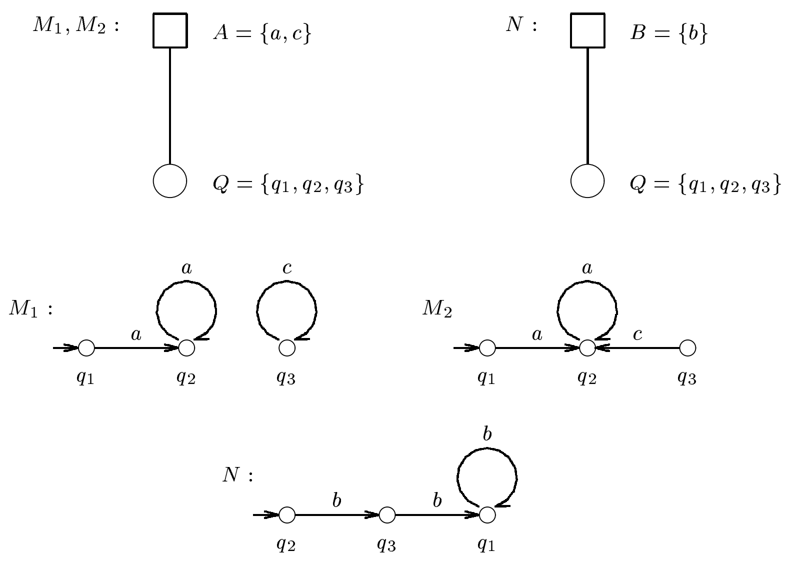


Figure 26: Agents M_1, M_2 and N.

Boris Trakhtenbrot
Automata and Hybrid Systems
Section 5: Monday 6 October 1997

Computing Science Department
Uppsala University
Sweden

Section 5: Oracles

Restorability for Product and Projection 57
Relativization wrt Oracle-Acceptors and Oracle-Transducers 58
Relativizing a Transducer wrt a Transducer 59
Relativization wrt Languages and Retrospective Operators 60
An Example: Timed Automata 60
Relativizing the Five Main Issues 62
Comments and Examples 62
Circuits: Operational versus Denotational Semantics 65
On Data Flow Networks 66

The main goal of this lecture is to define acceptors and transducers, which are equipped with oracles, and to discuss the way one can, or cannot extend for them The Five Issues. The considerations rely on synchronous nets, as the relevant interaction paradigm, and they make use of their restorability properties. Finally, we observe that, wrt circuits of transducers, restorability acquires the flavor of Kahn's Principle (which is nontrivial for Data-flow Networks).

Some abuse of notations/drawings for nets

Figure 27 shows a Petri graph of a net composed from agents $\mathcal{A}_1$, $\mathcal{A}_2$ with shared ports. A

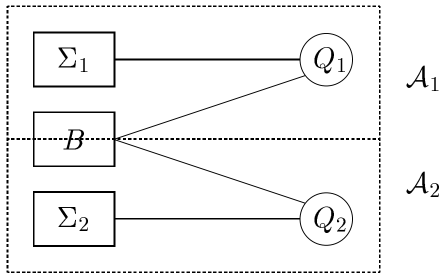


Figure 27: Petri Graph Example Notation 1

different notation is found in Figure 28 where agents are identified with their (unique) registers

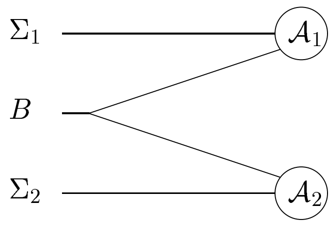


Figure 28: Petri Graph Example Notation 2

and the ports are sticking wires (the boxes are omitted). No distinction is made between the name of a port and the name of its alphabet; and similarly for registers.

Instead of circles, switching theorists use mnemonic geometric figures for boolean gates and the like. So why not draw like in Figure 29? Note that in this figure, boxes are not ports but boxes and circles in Petri graphs are still respected.

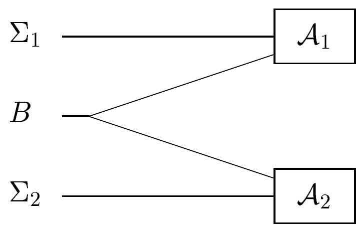


Figure 29: Petri Graph Example Notation 3

Restorability for Product and Projection

In the following, each agent T_i has one register; agents do not share registers; and product is interpreted as synchronous composition. If the T_i are equipped with anchoring (initialisation and fairness conditions) this should also be considered for the composition.

Definition: Let T_i $(i = 1, 2)$ be labelled transition systems with (respectively) state sets Q_i, alphabets $\Sigma_i \times B$, and transition relations $\rightarrow_i$. Their *product* $T_1 \times T_2$ is the labelled transition system T with

1. state set $Q = Q_1 \times Q_2$;
2. alphabet $\Sigma = \Sigma_1 \times \Sigma_2 \times B$; and
3. transition relation given by

$$\langle q_1, q_2 \rangle \xrightarrow{a_1,a_2,b} \langle q_1', q_2' \rangle \qquad \text{iff} \qquad q_1 \xrightarrow{a_1,b}_1 q_1' \text{ in } T_1 \quad \text{and} \quad q_2 \xrightarrow{a_2,b}_2 q_2' \text{ in } T_2.$$

The *product* of acceptors $\mathcal{A}_i = \langle T_i, \textsc{Init}(\mathcal{A}_i), \textsc{Fair}(\mathcal{A}_i) \rangle$ (for $i = 1, 2$) is the acceptor $\mathcal{A} = \langle T, \textsc{Init}(\mathcal{A}), \textsc{Fair}(\mathcal{A}) \rangle$ where

1. $T = T_1 \times T_2$;
2. $\textsc{Init}(\mathcal{A}) = \textsc{Init}(\mathcal{A}_1) \times \textsc{Init}(\mathcal{A}_2)$; and
3. $Q' \in \textsc{Fair}(\mathcal{A})$ iff its projections onto Q_1 and Q_2 are respectively in $\textsc{Fair}(\mathcal{A}_1)$ and $\textsc{Fair}(\mathcal{A}_2)$.

Remark: Note that the product of LTSs which are deterministic and complete need not be complete.

Definition: Let T be a labelled transition system over alphabet $\Sigma_1 \times \Sigma_2$. The *projection* of T onto Σ_1, *Project*(T, Σ_1), is the LTS over alphabet Σ_1 with the same set of states as T, and

$q \xrightarrow{a} q'$ in *Project*(T, Σ_1) if there is $b \in \Sigma_2$ such that $q \xrightarrow{a,b} q'$ is a transition of T. The projection of an ω-acceptor $\mathcal{A}$ is defined by taking the projection of its LTS. The initial states and fairness conditions of *Project*$(\mathcal{A}, \Sigma_1)$ are inherited from $\mathcal{A}$. We can depict a projection as in Figure 30.

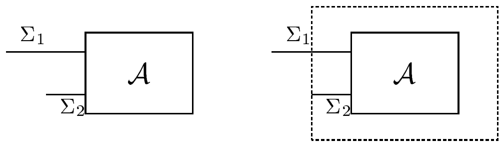


Figure 30: Projection

Remark: Note that if T is deterministic then its projection is not necessary deterministic. Hence the projection is not well defined on transducers.

Restorability Claims:

1. Let $R_i(x_i, y)$ be the characteristic function of the ω-language defined by ω-acceptors $\mathcal{A}_i$ over alphabet $\Sigma_i \times B$. Then $R(x_1, x_2, y) = R_1(x_i, y) \wedge R_2(x_2, y)$ is the characteristic function of the ω-language accepted by $\mathcal{A}_1 \times \mathcal{A}_2$.
2. Let $R(x, y)$ be the characteristic function of the ω-language defined by ω-acceptors $\mathcal{A}$ over alphabet $\Sigma_1 \times \Sigma_2$. Then $\exists y.R(x, y)$ is the characteristic function of the ω-language accepted by *Project*$(\mathcal{A}, \Sigma_1)$.

These two restorability claims are strong: they provide an explicit presentation for the restorable ω-language. Note the clear logical flavour of this presentation in terms of conjunction and $\exists$-quantification.

Relativization wrt Oracle-Acceptors and Oracle-Transducers

Let $\mathcal{A}$ be an acceptor over the alphabet $\Sigma_1 \times \Sigma_2$ and let $\mathcal{B}$ be an acceptor over alphabet Σ_2, as shown in Figure 31. The *relativization* $\mathcal{A}^{\mathcal{B}}$ of $\mathcal{A}$ with respect to $\mathcal{B}$ is defined as *Project*$(\mathcal{A} \times \mathcal{B}, \Sigma_1)$. An ω-language L is *defined* by $\mathcal{A}$ with respect to oracle $\mathcal{B}$ if L is defined by $\mathcal{A}^{\mathcal{B}}$. In this case we say also that L is *$\mathcal{B}$-definable* by $\mathcal{A}$.

Note that if $\mathcal{A}$ is Σ_1-deterministic then $\mathcal{A}^{\mathcal{B}}$ is deterministic. We say that L is *$\mathcal{B}$-definable* by a deterministic ω-acceptor if there exists Σ_1-deterministic acceptor $\mathcal{A}$ such that L is $\mathcal{B}$-definable by $\mathcal{A}$.

Remark: In the above, only a single occurrence of an oracle $\mathcal{B}$ is considered. The generalisation for several occurrences (copies) of the oracle is immediate but notationally cumbersome. Hence, we confine ourselves to the single occurrence version.

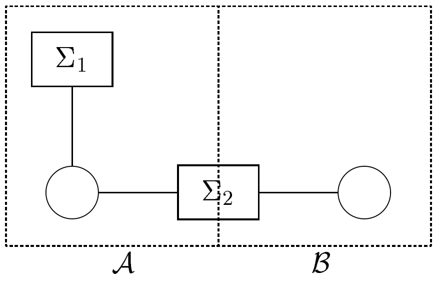


Figure 31: Relativization Example

Remark: Since a transducer $\mathcal{B}$ can also be considered as a (safe) acceptor (it accepts the graph of the corresponding operator) we know already what the relativization with respect to an oracle transducer is.

Relativizing a Transducer wrt a Transducer

Let $\mathcal{A} = \langle T_1, q_1 \rangle$ be a transducer of the type $\Sigma_1 \times \Sigma_4 \rightarrow \Sigma_2 \times \Sigma_3$ and let $\mathcal{B} = \langle T_2, q_2 \rangle$ be a transducer of type $\Sigma_3 \rightarrow \Sigma_4$ which computes a strongly retrospective ω-operator. Then $\mathcal{A}^{\mathcal{B}}$ is defined as $\langle T, q \rangle$, where T is *Project*$(T_1 \times T_2, \Sigma_1 \times \Sigma_2)$ and $q = \langle q_1, q_2 \rangle$ is the pair of initial states q_1 and q_2. Under the above assumption it can be shown that $\mathcal{A}^{\mathcal{B}}$ is a transducer of type $\Sigma_1 \rightarrow \Sigma_2$ and hence computes an ω-retrospective operator F. We say that F is *computable* by $\mathcal{A}$ with respect to oracle $\mathcal{B}$. This is depicted in Figure 32.

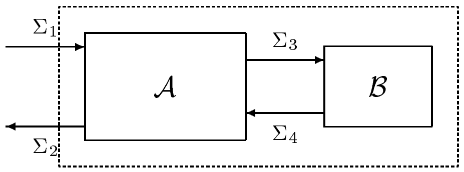


Figure 32: Transducer $\mathcal{A}$ relativized wrt transducer $\mathcal{B}$

Remark: Note that if the retrospective operator computed by $\mathcal{B}$ is not strongly retrospective then $\mathcal{A}^{\mathcal{B}}$ is not necessarily a transducer; "causal circularities" can appear. We will reconsider this point in the more general setting of "reliable feedback" (see page 69).

Claim (Congruence): If

1. $\mathcal{B}_1$ and $\mathcal{B}_2$ define the same ω-language; and
2. $\mathcal{A}_1$ and $\mathcal{A}_2$ define the same ω-language.

then $\mathcal{A}_1^{\mathcal{B}_1}$ and$\mathcal{A}_2^{\mathcal{B}_2}$ define the same ω-language.

Similarly for transducers: if

1. $\mathcal{B}_1$ and $\mathcal{B}_2$ are transducers that compute the same strongly retrospective operators; and
2. transducers $\mathcal{A}_1$ and $\mathcal{A}_2$ compute the same retrospective operator;

then $\mathcal{A}_1^{\mathcal{B}_1}$ and $\mathcal{A}_2^{\mathcal{B}_2}$ compute the same retrospective operator.

Relativization wrt Languages and Retrospective Operators

In the following, we shall focus on oracles which are strongly retrospective operators (motivated by the above remark).

Let $\mathcal{B}$ be a transducer that computes a strongly retrospective operator F; we say that an ω-language L' is *F-definable* by $\mathcal{A}$ is L' is $\mathcal{B}$-definable by $\mathcal{A}$. The congruence claim from above shows that the particular choice of $\mathcal{B}$ does not matter; and similarly for F-computability, etc.

L is F-definable by a finite state acceptor if there is a finite state acceptor $\mathcal{A}$ such that L is F-definable by $\mathcal{A}$.

The role of the congruence claim is to justify the robustness of the concepts above. The following theorem follows from our two restorability claims from page 58, and is crucial for the connection with logic.

Theorem: Let $\mathcal{A}$ be an ω-acceptor over alphabet $\Sigma_1 \times \Sigma_2$ and let $\mathcal{B}$ be an ω-acceptor over alphabet Σ_2. Let $A(x,y)$ and $B(y)$ be the corresponding characteristic functions of the ω-languages definable by $\mathcal{A}$ and $\mathcal{B}$. Then $\exists y.A(x,y) \wedge B(y)$ is the characteristic function of the language definable by $\mathcal{A}^{\mathcal{B}}$.

Remark: In the case of multiple occurrences of the oracle the formula would be

$$\exists y_1 \exists y_2 \ldots \exists y_k.A(x, y_1, y_2, \ldots y_k) \ \wedge \ B(y_1) \ \wedge \ B(y_2) \ \ldots \ \wedge \ B(y_k).$$

An Example: Timed Automata

Consider the $Z \times Y \to X$ transducer $\mathcal{A}$ in Figure 33, whose LTS has states $Q = \{wait, ring\}$ and three boolean alphabets X, Y and Z. The initial state is $wait$.

We want to relativize $\mathcal{A}$ as a Z-acceptor wrt a strongly retrospective $X \to Y$ operator F.

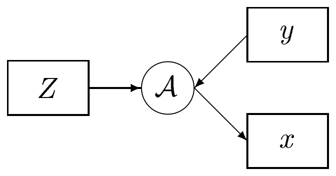
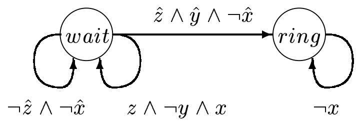


Figure 33: Timed Automaton



$wait$
$ring$
$\hat{z} \wedge \hat{y} \wedge \neg\hat{x}$
$\neg\hat{z} \wedge \neg\hat{x}$
$z \wedge \neg y \wedge x$
$\neg x$

Figure 34: States of $\mathcal{A}$

Let $\mathcal{B}$ be a transducer which computes F. Then according to our definitions we have to consider $\mathcal{A}^{\mathcal{B}} \overset{def}{=} Project[\mathcal{A} \times \mathcal{B}, Z]$.

In particular we could, as F, take the operator SqClock and a transducer *Trans*(SqClock) which computes it. SqClock-definability reminds definability by timed automata up to the following peculiarities:

1. Discrete time.
2. Type of clock constraints.

Actually, the operational semantics for timed automata as well as for hybrid automata is also based on the product of a finite state automaton and a clock (oracle), though this is not mentioned explicitly. We shall return to this in Lecture 6.

The states (residuals) of *Trans*(SqClock) can be identified with the current value displayed by the clock. Hence a state of $\mathcal{A} \times$ *Trans*(SqClock) is actually a pair $\langle q, val\rangle$.

For example:

$$\langle wait, value\rangle \xrightarrow{1,1,0} \langle ring.value + 1\rangle$$

Now, assume that at moment t, the current action of $\mathcal{A}$ is 1 (i.e., $\hat{z} = 1$) and val is a square number ($\hat{y} = 1$). Then the transition

$$\langle wait, val\rangle \xrightarrow{\hat{z},\hat{y},\neg\hat{x}} \langle ring.val + 1\rangle$$

could be executed with $\hat{x} = 0$, which means that no reset happens. The next state of $\mathcal{A}$ will be $ring$ and the next state of $\mathcal{B}$ will be $val + 1$.

Relativizing the Five Main Issues

Below F is a strongly retrospective ω-operator and $R(x, y)$ is the characteristic function of its graph.

The following two relativizations hold:

1. **Monadic logic** An ω-language L is F-definable by a finite state acceptor iff there is a monadic formula ψ such that $\exists x \exists y.\psi \wedge R(x, y)$ is the characteristic function of L.

2. **Nets** The set $\{F, \text{PE}^{\wedge}, \text{PE}^{\neg}, \text{DELAY}^{a}\}$ is a finite basis for the class of all retrospective ω-operators which are F-computable by finite state transducers.

The following three statements may hold or fail depending on the choice of oracle F. It is clear that they hold when F has finite state memory.

3. **Determinization** A language is F-definable by a finite state acceptor iff it is F-definable by a deterministic finite state acceptor.

4. **Uniformization** An ω-language L over alphabet $\Sigma_1 \times \Sigma_2$ which is F-definable by a finite state acceptor contains the graph of a retrospective operator iff L contains the graph of a retrospective operator which is F-computable by a finite state transducer.

5. **Decidability of emptiness** It is decidable whether the language F-definable by a finite state acceptor is empty.

Comments and Examples

1. The translation from Acceptor to some formula in Logic $L_2^{<,R}$ is straightforward, but we don't want simply a formula in $L_2^{<,R}$ and insist on the specific format $\exists x \exists y.\psi \wedge R(x, y)$. Transition from Logic to Acceptor is tricky and relies on the Büchi Theorem.

2. Here is where strong retrospectiveness of F is needed since it supports feedback reliability.

3.–5. For each subset of these three there exists an oracle which validates it and refutes the others.

More about the determinization issue

In Figure 35 M is a finite-state acceptor and F is an oracle. Recall that the acceptor is said to be **deterministic wrt** z **and** y iff $\forall \hat{z} \forall \hat{y} \forall \hat{q}$ there exists at most one $\hat{x}$ and at most one $\hat{q}'$ such that the transition holds, then the following crucial fact holds:

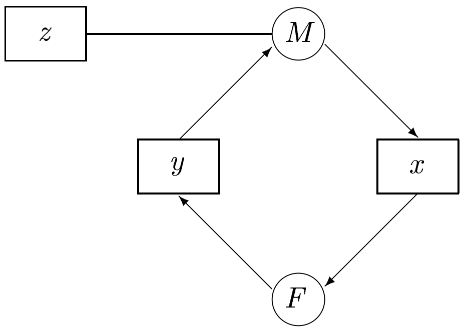


Figure 35: Acceptor M relativized wrt oracle F

Claim (Complementation): Let $L(z)$ be the ω-language which is F-definable by M (i.e. is accepted by M wrt F). Then the complementary finite-state acceptor ω-language $\neg L(z)$ is also F-definable by some finite-state acceptor.

Important Point: The third issue refers to just this kind of determinism.

If we would require simple determinism, i.e. $\forall\hat{z}\forall\hat{y}\forall\hat{q}\forall!\hat{x}\exists!\hat{q}'$ such that the transition holds, then the Complementation Claim could not be guaranteed.

Refuting determinization of relativized acceptors

Example 1 (Oracles generalizing the Square clock). Let $E \subset N$. Consider $ETimer$ - the associated retrooperator, which outputs a 1 at time t if exactly a number $\tau \in E$ of steps have passed since the input last registered a 1.

Now, assume that in Figure 36 M is a fixed finite-memory acceptor, whereas F may be an arbitrary $ETimer$.

Claim (easy): Can chose M so that for every E, the language L_E accepted by M relatively to $ETimer$, is as follows:

$$z \in L_E \ \ iff \ \ \exists m \exists n (z(m) = 1 \ \wedge \ z(n) = 1 \ \wedge m - n \in E.$$

Exercise: Show that for some appropriate E the language which is complementary to L_E is not acceptable by any finite memory acceptor relativized wrt $ETimer$. Or (equivalently,according to the Complementation Claim) show that for some E the acceptor M above cannot be replaced by a zy-deterministic acceptor.

Loop-free Relativization

This notion, which is implicit in a paper of Fix, Alur and Henzinger, is best illustrated by an example. Consider acceptor M with ports z, y and oracle Orc with input port z and output port y ; hence, M is affected by the output y of the oracle, but the oracle has no input from M.

Let L be the language accepted by M^{Orc} :

$$z \in L \quad \text{iff} \quad \exists y[M(z,y) \wedge y = Orc(z)] \qquad (*)$$

It follows that one can perform on M the usual determinization, without changing the accepted language, and still preserving Loopfree Relativization. This remark obviously implies the following facts about Loopfree Relativization.

(i) If L is Orc-acceptable, so is $\neg L$, the complement of L. Indeed, one can adapt for $\neg L$ the deterministic Orc-acceptor of L; just take the complementary fairness conditions. But

Warning. This has nothing to do with closure under hiding (quantification). It still can happen that some $L(z_1, z_2)$ is Orc-definable via loopfree relativization but $\exists z_2 L(z_1, z_2)$ is not. (Note the instructive contrast with non-deterministic Buchi Automata, which supports closure under hiding, but cannot be determinized with Buchi fairness conditions).

(ii) Orc-definability with loopfree relativization implies Orc-definability (in the initial sense! – explained before the Complementation Theorem) by zy deterministic acceptor.

Refuting Emptiness Decidability

Example 3. Associate with $E \subseteq N$ the retrooperator f_E: $y(t) = 1$ iff $t \in E$ (hence,a constant operator, no dependence on the input x).

Example 4. With each $n \in N$ associate the acceptor A_n for the language $L_n(z,y) =$ def $z[0,n) = 0$ and $z(n) = 1$ and $y(n) = 1$.

Claim. $n \in E$ iff $A_n^{f_E}$, the f_E-relativization of A_n, accepts a nonempty language.

Obviously, decidability of E is reduced to emptiness decidability for $A_n^{f_E}$.

Refuting Uniformization

Associate with an arbitrary retrooperator f of type $X_1 \Rightarrow X_2$ the retrooperator F of type $X_1 \times X_2 \Rightarrow Boolean$, as follows:

$b(t) = 1$ iff $x_2[0,t) = f(x_1[0,t))$.

Say also that f is the explicit transform of F.

Claim 0 (Properties of F). (i) F is strongly retrospective; (ii) The output of F stabilizes, i.e. $\forall x_1 x_2[y = F(x_1, x_2) \rightarrow \exists t \forall n(n > t \rightarrow y(n) = a)]$ where a is one or zero.

Consider the language L:

$L(s_1, s_2) = \text{def}\{s_1 s_2 | F(s_1, s_2) = \lambda t.1\}$

Claim 1 (Properties of L). (i) L is the graph of f; (ii) L is F-definable by a finite-memory acceptor A.

Proposition (About f which is not F-computable). *There exist retrooperators f with the following property: Let F be the implicit transform of f, and let h be the retrooperator which is F-computed by an arbitrary finite-memory transducer H; then h differs from f.*

Assume that f is as in the proposition above, and that L is the graph of f. Then it is straightforward to check that L and F refute the Uniformization Issue.

Circuits: Operational versus Denotational Semantics

Here we present for synchronous nets of transducers the flavour of Kahn's Principle which is known for DataFlow Networks.

Example 1

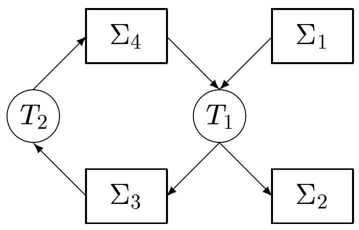


Figure 36: A Net of two transducers

Consider the net in Figure 36. The net contains two transducers T_1 and T_2.

T_1 *has inputs x_1 and x_2, and outputs y_1 and y_2. T_1 computes a retrospective operator $y_1 y_2 = F_1(x_1, x_2)$.*

T_2 *has input y_1, and output x_1. T_2 computes a retrospective operator $x_1 = F_1(y_1)$.*

The corresponding system, Syst, of equations has the same picture with F_i staying instead of T_i.

Questions:

a) Has *Syst* an unique solution H (i.e. $x_1 y_1 y_2 = H(x_2)$)?

b) Is $T_1 \times T_2$ a transducer from X_2 to $X_1 \times Y_1 \times Y_2$ and hence computing a retrospective operator $x_1 y_1 y_2 = G(x_2)$

Theorem: Under some conditions (see below “feedback delay”, i.e. no “circular causality”) the answers to the questions above is yes and $G = H$.

Feedback Delay

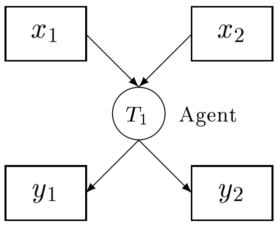


Figure 37: Feedback Delay

The agent T_1 in Figure 37 is supposed to be a transducer with $X = X_1 \times X_2$ and $Y = Y_1 \times Y_2$, and with functions $next : Q \times X \to Q$ and $out : Q \times X \to Y$, such that $q \xrightarrow{\hat{x},\hat{y}} q' \Leftrightarrow q' = next(q, \hat{x})$ and $\hat{y} = out(q, \hat{x})$.

Actually $\hat{y}_1 = out_1(q, \hat{x}_1, \hat{x}_2)$ and $\hat{y}_2 = out_2(q, \hat{x}_1, \hat{x}_2)$.

Definition: y_1 is *delayed* wrt x_2 iff $y_1 = out_2(q, \hat{x}_1)$ i.e. $\hat{x}_2$ is dummy. Otherwise y_1 is a subordinate to x_2.

Definition: A feedback in a $\times$-net (circuit) is *delaying* if it passes through at least one delaying link. The condition of the Theorem is that each feedback of the net is delaying.

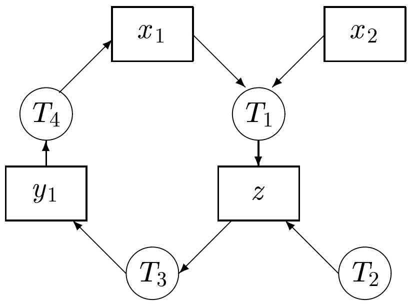


Figure 38: Delaying Feedback

In Figure 38 the port z of T_1 is supposed to be delayed wrt x_1 (though it may happen that it is subordinated to x_2.

On Data Flow Networks

Data flow networks are a special kind of asynchronous nets, whose components perform stream processing. Under appropriate conditions each component computes (in a very specific sense) a stream operator. As in the case with (synchronous) nets, a system of equations $Syst$ is associated with the net N and questions a)–b) from above suggest themselves. G. Kahn's theorem gives the positive answers for both questions but the proof is not so straightforward as for circuits; the asynchronous interaction is more subtle than the synchronous one.

Boris Trakhtenbrot
Automata and Hybrid Systems
Section 6: Monday 20 October 1997

Computing Science Department
Uppsala University
Sweden

Section 6: Lifting to Hybrid Systems

Reliable Feedback 69
Duration Labelled Transition Systems 71
Duration LTS Machinery for Retrospective Operators 73
Relativization wrt Retrospective Operators 74
Relativizing the Five Issues in Continuous Time 75
Reducibility Among Oracles 76
Discussion 77

In this Lecture we lift the notions and results of Lecture V from discrete time and ω-sequences to continuous time. Note, however, that instead of considering non-Zeno signals (as in Lectures II and III) here we confine ourselves to only right-continuous ones. This makes the exposition simpler (avoiding some annoying technicalities), but still preserves the essential features of the subject. We formalize the intuition of Hybrid Systems (HS) for right-continuous signals, similarly to the discrete-time case, as a relativized acceptor or relativized transducer, i.e., according to the formula:

$$A^B = project(A \times B;\ \Sigma).$$

For this sake we have first to explain for continuous time the meaning of the components A, B and of the operations (synchronous composition and projection).

Note, that A is a finite-state (hence, a speed-independent) acceptor or transducer, as explained already in Section 3. However, we still have to extend the *LTS* machinery to continuous time, in order to appropriately handle oracles B with infinite, even uncountable, memory. Their composition and projection can be handled exactly as in the discrete case.

At the other hand, causal circularities in nets over continuous time are more subtle than in the discrete case. This is shown in the subsections about reliable feedback and about reducibility among oracles.

We conclude by pointing out some terminological/notational discord in the area.

Reliable Feedback

For nets over ω-retrospective operators the following is easy:

Proposition: **(Reliable feedback)** If every cycle in a net N contains a strongly retrospective operator then N is well-defined, i.e., has an unique solution.

However, for nets over signal retrospective operators this claim is wrong, and therefore we are looking below for conditions which guarantee well-definedness.

Proposition: Let N be a net over retrospective operators, each of them being stable (in particular speed independent) or with fixed variability. If every cycle of N contains an operator, which is strongly retrospective and has fixed variability, then N is well defined (i.e., has a unique functional solution in the class of all retrospective operators).

According to Table 4, this Proposition is applicable to PTIMER and to FILTER but not to TIMER nor to DELAY^a. In fact, for TIMER there holds the following negative result.

Proposition: There exists a net over pointwise operators and TIMER which contains a TIMER in every cycle, but is not well defined.

Nevertheless, for DELAY^a there holds.

Proposition: Let N be a net over retrospective operators. If each cycle in N contains a DELAY^a, then N is well defined.

The Effect of Delaying on Well-Definedness (Reliability)

F is a *delaying* operator iff for all x and $t > 0$ there is $\delta > 0$ such that $F(x)(t)$ depends only on $x[0, t - \delta)$. F is a *uniformly delaying* operator iff there is $\delta > 0$ such that for all x and $t > 0$, $F(x)(t)$ depends only on $x[0, t - \delta)$.

A particular case of a delaying (but not uniformly delaying) operator is the ψ-decreasing delay operator $D_\psi(x)(t) = x(t - \psi(t))$ where ψ is a decreasing positive-valued function with $\lim_{t\to\infty} \psi(t) = 0$.

Exercise: Show that the final Proposition above holds for arbitrary uniformly delaying operators. Also, show it holds for ψ-decreasing delay, but it fails for some delaying operator.

(Hint: Consider the input-less net in Figure 39. We are looking for the unique fixed

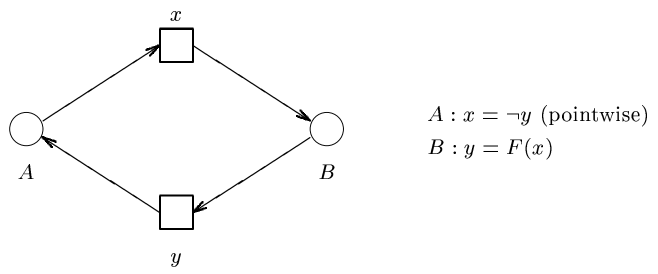


Figure 39: An input-less net.

point $x = \neg F(x)$ which should be a non-Zeno signal. Now, see what kind of delaying can guarantee this.)

Delaying in the sense of the circuit paper (Def. 6, p. 7)

Proposition: There is no net over pointwise signal operators and DELAY, TIMER, PTIMER and FILTER that defines NMDELAY.

Duration Labelled Transition Systems

Labelled Transition Systems (*LTS*s) are intended to be an executable formalism with operational semantics. A transition may be executed when it is enabled, and consecutive executions of transitions generate runs which underly the semantics of acceptors and transducers. The focus in previous lectures was on speed-independent acceptors and transducers; hence there were no explicit restrictions on the duration of the lasting steps. Since we want also to cover speed dependence, the duration of transitions should be reflected somehow in the syntax. These remarks suggest the following (nonconstructive) "syntax" of *Duration LTSs*, which are appropriate for processing right-continuous signals:

Duration Transitions over alphabet $\Sigma = \{a, \dots\}$ have the format

$$M(q_1, a/\delta, q_2)$$

where δ is a positive real denoting the *duration* of $M(q_1, a/\delta, q_2)$.

Note: Without loss of generality we assume that $0 < \delta \leq 1$; in this way some annoying technicalities can be avoided.

(i) $M(q_1, a/\delta, q_2)$ is enabled at instant t toward $t + \delta$ iff $q(t) = q_1$ and $x[t, t + \delta) = a$:

(ii) Execution: $q(t + \delta) = q_2$.

Let M be a duration *LTS*. A run of M is an infinite sequence

$$q_0 a_0 \delta_0 q_1 a_1 \delta_1 \cdots$$

such that for all i, $M(q_i, a_i/\delta_i, q_{i+1})$ holds. Its duration is $\delta = \sum(\delta_i)$. It is a Zeno run if δ is finite; otherwise it is a non-Zeno run, and we say that it supports the right-continuous signal $a_0^{\delta_0} \cdot a_1^{\delta_1} \cdots$. (The notation a^δ stands for the signal fragment which is defined on $[0, \delta)$ and has the constant value a.)

If M is anchored then (as usually) we consider the corresponding language L it defines.

Definition: ($\times$-composition, synchrony) As usual:

$$M(p_1 q_1, \langle a, b, c\rangle/\delta, p_2 q_2) \overset{def}{=} M_1(p_1, \langle a, b\rangle/\delta, p_2) \quad \text{and} \quad M_2(q_1, \langle b, c\rangle/\delta, q_2).$$

Definition: (The join of languages) As usual.

Definition: (Density of M) If $M(q_1, a/\delta, q_2)$ holds and $\delta = \delta_1 + \delta_2$ then there exists $q_{1.5}$ such that $M(q_1, a/\delta_1, q_{1.5})$ and $M(q_{1.5}, a/\delta_2, q_2)$.

Proposition: (Restorability) If M_1, M_2 meet the density requirement and don't share registers then $L(M_1 \times M_2) = L(M_1)$ join $L(M_2)$.

In the sequel density is always assumed, and $\times$ is applied only to nets (no sharing of variables/registers).

Let T be a duration *LTS* over the alphabet $\Sigma_1 \times \Sigma_2$.

Definition: T is a *duration transducer of type* $\Sigma_1 \longrightarrow \Sigma_2$ iff the following three conditions hold:

1. **Functionality:** There exist partial(!) functions $\mathit{next} : Q \times \Sigma_1 \times (0,1] \longrightarrow Q$ and $\mathit{out} : Q \times \Sigma_1 \times (0,1] \longrightarrow \Sigma_2$ such that $T(q_1, \langle a, b\rangle/\delta, q_2)$ holds iff $\mathit{next}(q_1, a, \delta) = q_2$ and $\mathit{out}(q_1, a, \delta) = b$
2. **Duration:** There is a total function $\mathit{maxdur} : Q \times \Sigma_1 \longrightarrow (0,1]$ such that $\mathit{maxdur}(q,a)$ is the maximal δ for which $\mathit{next}(q,a,\delta)$ and $\mathit{out}(q,a,\delta)$ are defined.
3. **Zeno avoidance:** Every sequence

$$q_1 a \delta_1 q_2 a \delta_2 q_3 \cdots$$

   where $q_{i+1} = \mathit{next}(q_i, a, \delta_i)$ and $\delta_i = \mathit{maxdur}(q_i, a)$ is a non-Zeno run, i.e., $\sum \delta_i = \infty$.

Proposition: If T is a duration transducer of type $\Sigma_1 \longrightarrow \Sigma_2$ then the language it accepts under safe anchoring is the graph of a retrospective operator $F(T)$ of type $\Sigma_1 \longrightarrow \Sigma_2$. We say that T *defines (computes)* the operator $F(T)$.

Note: Actually, given a Σ_1 signal ξ as input, the "computation" of the corresponding output Σ_2-signal η can be performed via (consecutive) steps as follows: Assume that after k executions:

(i) The input fragment $\xi[0,t)$ has been processed;

(ii) $\xi(t) = a$;

(iii) $q(t) = q'$;

(iv) the next discontinuity of ξ after t is at $t + \rho$; and

(v) $\delta' = \min(\rho, \mathit{maxdur}(q', a))$.

Then the $(k+1)$-th step is: $\mathit{next}(q', a, \delta')$ and $\mathit{out}(q', a, \delta')$.

Duration LTS Machinery for Retrospective Operators

In general, with a given retrospective operator F of type $\Sigma_1 \longrightarrow \Sigma_2$ we can associate a transition system *Trans*(F) as follows:

States $\longleftrightarrow$ Residuals of F.

(a) Format of transitions:

$$T(q_1, \langle\phi, \psi\rangle, q_2)$$

with input fragment $\phi : [0, \delta) \longrightarrow \Sigma_1$ and output fragment $\psi : [0, \delta) \longrightarrow \Sigma_2$ (for some positive "duration" δ).

- (b) $T(q_1, \langle\phi, \psi\rangle, q_2) \in Trans(F)$ iff whenever q_1 is subjected to the input fragment ϕ it outputs the fragment ψ and reaches the residual q_2.

But since we consider only operators over right-continuous signals, we can confine ourselves to transitions in which both ϕ and ψ are constant on $[0, \delta)$.

Proposition: Let F be a right-continuous retrospective operator. Then $Trans(F)$ is a $\Sigma_1 \longrightarrow \Sigma_2$ duration transducer which computes F.

Example 1. The speed independence case

Assume F is speed independent (and hence with at most countable memory). Then no restrictions are imposed on the durations, and we can therefore manage with a "bare" syntax in which the duration information is omitted. That is exactly the case considered in Section 3, except that density was not assumed there. The reason: at that stage no interaction and/or restorability issues for acceptors were on the agenda. The density postulate could be added there, without affecting the results of Section 3.

Example 2. Timer

Consider the retrospective operator TIMER and the corresponding transducer *Trans*(TIMER).

We present an encoding of the states and a symbolic description of the duration transitions, relying on the representation of TIMER through actuators and sensors (see page 24).

$$y \;=\; \textsc{Timer}(x) \;=\; \textsc{Sensor}(\textsc{Actuator}(x)).$$

The residual of TIMER at moment t is fully recovered from the pair:

$$\begin{aligned} elapse(t) &\overset{def}{=} \lim_{\varepsilon \to 0} z(t - \varepsilon) \\ \alpha(t) &\overset{def}{=} \lim_{\varepsilon \to 0} x(t - \varepsilon) \end{aligned}$$

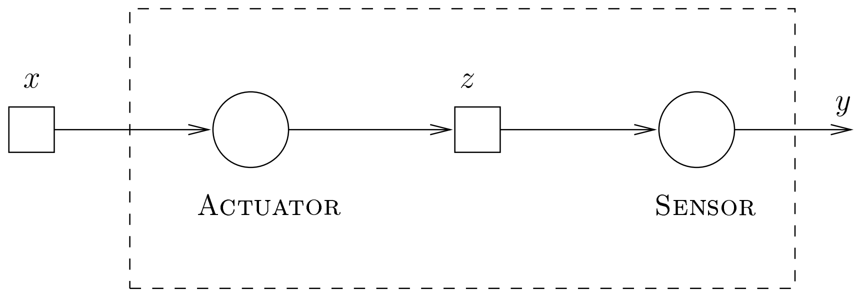
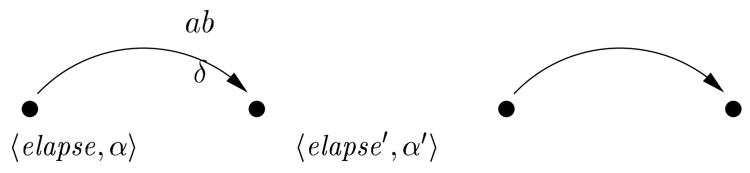


Figure 40: Schematic description of Timer

Hence, the states of *Trans*(Timer) are identified with pairs $\langle elapse : \text{Real},\ \alpha : X\rangle$

The transitions will have the format:


ab
δ
$\langle elapse, \alpha\rangle$
$\langle elapse', \alpha'\rangle$

where:

- $\alpha' = a$;
- $\alpha = a$ (no reset) $\implies$ $elapse' = elapse + \delta$;
- $\alpha \neq a$ (reset) $\implies$ $elapse' = \delta$;
- $b = 1 \iff a = \alpha \wedge elapse \geq 1$;
- $elapse < 1 \implies \delta \leq 1 - elapse$.

Relativization wrt Retrospective Operators

Like in the discrete case:

$$A^F \overset{def}{=} project\Big(A \times Trans(F); \Sigma\Big).$$

Note the fundamental asymmetry between the parameters:

- A has finite memory, and hence is a constructive object.
- F has in general infinite memory, hence $Trans(F)$ is in general non-constructive.

This means that the available syntax is that of A, whereas the oracle (black box) F participates only in the semantics. When and how A can execute a transition from instant t toward $t+\delta$ depends on how the oracle behaves on this semi-interval of time. This approach was already illustrated wrt the (discrete time) oracle SqClock, without explicit reference to the above definition of A^F (see page 60).

As a consequence, natural problems (notably algorithmic ones) assume a fixed (kind of) oracle F, and deal with finite relativized F-agents. For example, *Timed Automata* [3] actually use the oracle Timer.

Assume that A is a duration transducer. Already in the discrete case we have seen that in order to guarantee that A^F will also define a retrospective operator, additional restrictions have to be imposed on F; namely, F was assumed to be strongly retrospective. In the continuous case this does not suffice.

But from previous Propositions the following immediately follows.

Proposition:

(a) Assume that the oracle F is strongly retrospective and has fixed variability; then for every duration transducer A the relativization A^F computes a retrospective operator.

(b) Similarly in the case when F is a uniformly delaying operator.

Question: Is A^F actually a duration transducer?

Relativizing the Five Issues in Continuous Time

Below, F is a feedback reliable signal operator over right-continuous signals, and $R(x,y)$ the characteristic function of its graph. Monadic formulas from $L_2^<$ are interpreted in the structure of right-continuous signals.

1. **Monadic Logic** A signal language L is F-definable by a finite-state acceptor $\mathcal{A}$ iff there is a monadic formula ψ such that $\exists x \exists y. \psi \wedge R(x,y)$.
2. **Nets** The set $\{F, \mathrm{PE}^{\wedge}, \mathrm{PE}^{\neg}, \mathrm{NMDelay}\}$ is a finite basis for all retrospective signal operators which are F-computable by finite-state transducers.

3. – 5. These may or may not hold, depending on the choice of oracle F.

Remark: It is worth comparing the comments about strong retrospection and determinism in the discrete case (page 62) with their counterparts for continuous time. There, the oracle F is strong retrospective; here, feedback reliable. In the discrete case, determinism wrt z and y requires universal quantification over z and y. In the continuous case, in addition, δ should also be universally quantified. In both cases this kind of determinism implies the Complementation Claim (page 63).

Remark: The metricals PTimer, Filter and Delay^a are feedback reliable; hence, the five issues hold when either of these is taken as oracle F. This argument is not applicable to Timer. Recall that finite agents with oracle Timer are essentially the same as Timed Automata [3]. So, what about the five issues for them?

1. **Monadic logic** Still holds.
2. **Nets** Still holds.
3. **Determinization** Fails.
4. **Uniformization** Fails.
5. **Decidability of emptiness** Holds. This is the result of Alur and Dill [3] which underlies the verification method for timed automata and their applications.

In the literature, nets and transducers are not considered.

Reducibility Among Oracles

Is there among the metricals at hand a most powerful one? If so, then it might be a candidate to form (together with pointwise operators and with NMDelay) a preferred set of primitives. Of course, in the case of equal expressive power, extra arguments may be taken into consideration. We analyze expressive power in terms of finite memory reducibility and provide answers to these questions.

Operator F is *finite memory reducible* to operator G (notations $F \leq_{\text{fm}} G$) iff F is definable by a net over finite memory retrospective operators and G.

It is clear that $\leq_{\text{fm}}$ defines a preorder on the set of retrospective operators. We say that F and G have the same finite memory degree iff $F \leq_{\text{fm}} G$ and $G \leq_{\text{fm}} F$. We say that F and G are *incomparable* iff neither $F \leq_{\text{fm}} G$ nor $G \leq_{\text{fm}} F$.

Proposition: (Finite Memory Degrees)

1. Filter, Timer and PTimer have the same finite memory degree.
2. Delay and PTimer have incomparable finite memory degree.

It is instructive to compare the above Proposition with the following one which compares PTimer and Delay under assumptions on the variability of signals.

Proposition: (Equivalence of Delay and PTimer over signals with fixed variability)

1. For every k there exists a net N_k over finite memory retrospective operators and DELAY such that the operator defined by N_k coincides with PTIMER for k-variability signals.
2. For every k there exists a net N^k over finite memory retrospective operators and PTIMER such that the operator defined by N^k coincides with DELAY for k-variability signals.

Remark: The nets N_k and N^k in the above proposition contain a strongly retrospective delayed operator in every cycle.

Recall that if x has δ-latency and if $k \geq 1/\delta$ then x has k-variability. Therefore, by the above Proposition we have that DELAY and PTIMER have the same degree over the signals with any fixed positive latency.

Restricting Topology. An operator F is explicitly finite memory reducible to G (notation $F \leq^{\text{ex}}_{\text{fm}} G$) iff F is definable by a cycle free net over finite memory retrospective operators and G. Note that every cycle free net is well defined. Actually the function defined by such a net can be explicitly represented by composition.

Proposition: TIMER $\leq^{\text{ex}}_{\text{fm}}$ FILTER $\leq^{\text{ex}}_{\text{fm}}$ PTIMER.

(The opposite directions are unknown, but conjectured not to hold.)

Hence, PTIMER is preferable over TIMER because it ensures reliable feedback. It is also preferable to FILTER in view of the last proposition.

Discussion

In this section we provide a brief comparison with [11]. In this paper,

Deterministic Latency Delay (Definition 3) is similar to our FILTER; it is an operator.

Non Deterministic Delay (Definition 4) is actually a *language*, which is uniformed by FILTER.

Circuit (Definition 5) is a system of inclusions, on the form

$$x_i \in D_i(\cdots) \quad i = 1, 2, \ldots k.$$

unlike a system of equations in our approach. Nevertheless this system is referred to as a "system of equations" and needless to say, it need not have a unique solution. In actual fact, it is a system of languages L_i, and its semantics is the join of the components.

Timed Automata (Definitions 6 to 10) are heavily equipped with syntactic/pragmatic details.

The main results of the paper are as follows.

1. For each i there exists a timed automaton A_i which accepts the component language L_i.
2. There is a timed automaton A which accepts the join of the L_is.
3. Reasoning about systems using verification techniques for timed automata.

Warning: In [11] the model under consideration is called (like in many other sources) "asynchronous circuits". Note, however, that according to the classification adopted in Lecture IV, the composition in [11] is synchronous; the "asynchrony" means rather nondeterminism, which is contrasted with the determinism of operators. In Lecture VII we will discuss this kind of "asynchrony" in connection with McNaughton's work [13].

Boris Trakhtenbrot
Computing Science Department
Uppsala University
Sweden

Automata and Hybrid Systems
Section 7: Monday 27 October 1997

Section 7

Part 1. Faithful Principles of Timing 80

Part 2. Concluding Remarks 83

Challenges and Slogans 83
Main Features of the Theory 85
History and Problems 85
Acknowledgements 87

The first part of this lecture presents a short survey and discussion of Mc.Naughton's work, dated 1964, which, unfortunately, was never published. This work offers an elegant analysis of faithful timing principles, and is still inspiring for problems of today, concerning continuous time systems.

The second part is of concluding character.It summarizes and recapitulates challenges and features of the theory, open problems and historical information.

Part 1. Faithful Principles of Timing

McNaugton, on what is unrealistic in earlier theories.

> The theory of logical nets is an abstract switching theory concerning ideal elements put together to form ideal nets that behave according to ideal specifications. The problem of timing in these circuits? The theory is based on abstract principles, rather than on the property of specific hardware implementation; its validity transcends technological developments as well as those that are not anticipated.
>
> Unrealistic (with regard to certain physical principles) features of the early theories:
>
> a) switching elements that could switch *simultaneously* along with delays that would transmit a signal in *exactly* one unit of time.
>
> b) elements with arbitrarily large number of inputs and outputs (arbitrarily large fan-in and fan-out).
>
> c) time is synchronous and every element takes a *precisely* predictable amount (possibly zero) of time to operate.
>
> Five "realistic" stipulations:
>
> (S1) Upper bounds on fan-in and fan-out.
>
> (S2) Upper bound on the number of elements in a given volume.
>
> (S3) Upper bound on the distance between directly connected elements.
>
> (S4) Lower bound, greater than 0, on the operation time.
>
> (S5) The timing of any element is *not* precisely predictable.

What would be realistic assumptions and what are the corresponding problems?

> Assumptions about ε: There exists an ε such that for every δ, any process that reputedly takes δ units of time will actually take between $(1-\varepsilon)\delta$ and $(1+\varepsilon)\delta$ units of time.
>
> Assumptions about λ: There is a constant λ such that any element that reacts to a signal will only react if that signal is at least λ time units long.

The speed independence philosophy: We can never usefully and safely predict anything about how long any process will take.

The problem: Given the assumptions about λ and ε, can elements be designed so that, for some constant k, every finite-state event is represented by a net made up of such elements whose inways are to receive arbitrarily many signals at the rate of one signal per k units of time?

About delays:

- Compare with our previous remarks about finite/fixed variability and reliable feedback.
- Analyzing delay $\Rightarrow$ pointing on timing problems for (the simplest) identity function.

Simple delay: whatever the input is at time t, the output is the same at some later time t'. Formally:

(SD1) t' is continuous and strongly monotonic increasing in t;
(SD2) $t+1-\varepsilon \le t' \le t+1+\varepsilon$

The undesired feature of a chain of simple delays is that even if the input signal has no short steps *the output signal can get arbitrarily short steps* and, even worse, *unbounded variability.* Hence, violation of the assumption about λ, which forces us to insist that such a chain must keep signals, and intervals between them, above some minimal length.

Example: Consider a discrete finite-state operator F. Assume:

$$\begin{aligned} x &= x_1 x_2 \cdots x_n \cdots, \qquad \text{and} \\ F(x) = y &= y_1 y_2 \cdots y_n \cdots . \end{aligned}$$

We want an implementation of a right-continuous signal operator F' such that, for

$$\begin{aligned} x &= x_1^{\Delta t_1} x_2^{\Delta t_2} \cdots x_n^{\Delta t_n} \cdots \qquad \text{with } \textstyle\sum \Delta t_i = \infty, \\ F'(x) &= y_1^{\Delta t_1} y_2^{\Delta t_2} \cdots y_n^{\Delta t_n} \cdots . \end{aligned}$$

A weaker requirement confines to the implementation of an operator F^Δ, which is a "*slowed version*" of F'; that is, F^Δ applied to

$$\xi = a^\Delta x_1^{\Delta t_1} a^\Delta x_2^{\Delta t_2} \cdots a^\Delta x_n^{\Delta t_n} \cdots$$

returns

$$\eta = a^{\Delta} y_1^{\Delta t_1} a^{\Delta} y_2^{\Delta t_2} \cdots a^{\Delta} y_n^{\Delta t_n} \cdots ,$$

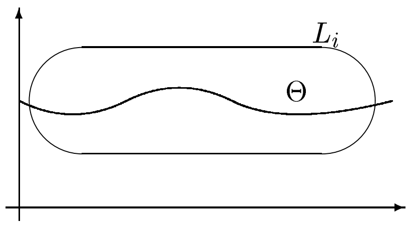
where a is an additional "idling" symbol. Even this is impossible; we would be happy even with an implementation that return an unpredictable output η' for a given ξ, as long as they all meet the requirement:

$$\eta' = a^{\delta_1} y_1^{\delta t_1} a^{\delta_2} y_2^{\delta t_2} \cdots a^{\delta_n} y_n^{\delta t_n} \cdots, \qquad \text{and}$$

$$\forall n \left\{ \tau_n \overset{def}{=} \textstyle\sum_{i \le n} \delta_i + \sum_{i \le n} \delta t_i < \mathcal{C}_1 + \mathcal{C}_2 t_n \right\} \qquad \text{for some constants } \mathcal{C}_1,\, \mathcal{C}_2.$$

Is this possible? Yes!

McNaughton's ε-λ-primitives are not operators, but rather languages.



$\{L_i\}$ a basis

L_i computes, in an unpredictable way, any function that uniformizes L. Hence, a synchronous composition of such primitives (actually the join L of some languages) may also specify each reasonable uniformizer of L, if such uniformizers exist.

Remember [11] (page 77) where existence/uniqueness of solutions is considered for a system of inclusions, instead of a system of equations.

Theorem: *(McNaughton)* For given λ,ε-requirements, there exist:

1. a basis $\mathcal{B}$ of ε-λ-blurred primitives, and
2. constants Δ, $\mathcal{C}_1$ and $\mathcal{C}_2$

such that for every discrete finite-state operator F, there exists a circuit over the basis $\mathcal{B}$ that computes F^{Δ} with constants $\mathcal{C}_1$, $\mathcal{C}_2$.

Comments

1. The theory handles *time approximation* as contrasted with value approximation in traditional settings.
2. The McNaughton primitives can be specified by timed automata, that use the *precise* oracle Timer. But the use of Timer as a primitive is not allowed, unlike in our theory of circuits. Analogy: In a team the primitives are human beings and not the cells these persons are built from; we cannot assemble some kind of Frankenstein monster. Hence the McNaughton skepsis is directed against precise operators in the frame of a realistic model of hardware and not in the frame of a specification formalism.
3. Here, as in [10], synchrony and asynchrony are not used in the sense of interaction combinators (as in our dichotomies). In the hardware community,"synchrony" often means the same as "precisely predictable amount of time", while "asynchrony" means just the opposite.

Part 2. Concluding Remarks

Challenges and Slogans

Hybrid systems, and in particular timed automata, are attractive paradigms of "reactive" systems. See our references, and further bibliographic pointers, to the extensive literature in the area. Important results for many versions of this paradigm and formalisms for their specification are reported there. These results support the solution of practical problems concerning:

1. Analysis and synthesis of hybrid systems.
2. Verification, in particular via model checking, of such systems.

Sometimes the theoretical results have a negative flavour, such as undecidability. Regrettably, there is still a lack of a coherent conceptual, terminological and notational setting even for the facts already known.

What are the methodological and expository embarrassment in this developing area? Here are two characteristic quotations:

> "The number of formalisms that purportedly facilitate the modeling, specifying and proving of timing properties for reactive systems has exploded over the past few years. The authors, who confess to have added to the confusion by advancing a variety of different syntactic and semantic proposals, feel that it would be beneficial to pause for a second – to pause and look back to sort out what has been accomplished and what needs to be done. This paper attempts such a meditation by surveying logic-based and automata-based real-time formalisms and putting them into perspective." [5]

"A new class of systems is viewed by many computer scientists as an opportunity to invent new semantics. A number of years ago, the new class was distributed systems. More recently, it has been real-time systems. The proliferation of new semantics may be fun for semanticists, but developing a practical method for reasoning about systems is a lot of work. It would be unfortunate if every new class of systems required inventing a new semantics, along with proof rules, languages and tools.

Fortunately, these new classes do not require any fundamental change to the old methods for specifying and reasoning about systems" [1]

The moral is that the explosion of models, definitions and notations should be cured by a better mathematical perspective on the theoretical aspects of the area even if its relevance to practice is not visible at this stage and may seem controversial.

As guiding methodological principles for dealing with new classes of problems, we pointed to the following:

- the separation of concerns and priorities; and
- not to ignore old-fashioned recipes.

Slogans

1. About concepts and tools: Automata theory and logic have priority over calculus. Hence, signals are over continuous time but take discrete values. No differential equations etc.

2. Semantics have priority over syntax. First we identify and characterize the basic semantic domains as a guide for appropriate syntax. Hence, languages and operators (the semantic universe) have priority over acceptors and transducers (the specification formalisms).

   The semantic universe is characterized in terms of: retrospection, residuals, speed independence, stability, etc. The main data structure is non-Zeno signals. Particular cases include burst signals and right-continuous signals.

3. Operators and transducers present an alternative option to languages and acceptors. Operators and transducers reflect two basic ideas:

   (a) Input/Output. Essential for:
      - Directed flow of information.
      - Communication: The output of one component feeds the input of another.
      - Learning via experiments with black boxes.
      - Strategies and games.

   (b) Determinism. Avoiding ambiguity. Precise predictability. But according to McNaughton [13] that is not a realistic principle even when theoretically possible.

4. Three kinds of specification formalisms:

   Declarative

   Our favourite choice is monadic logic $L_2^<$, which is believed to be *the* temporal logic.

   Executive

   Our favourite choice is bare syntax for labeled transition systems as acceptors and transducers.

   Interactive

   Our favourite choice is synchronous interaction embodied in nets of transducers and systems of functional equations. (See the "Kahn-like" theorem on page 66.)

5. Expressiveness has priority over algorithms, which in turn has priority over complexity.

Main Features of the Theory

1. Definitional: Our approach is guided by the intuitive slogan that a Hybrid System (HS) is a finite memory (and hence a speed-independent) agent with an oracle, both operating in continuous time and interacting synchronously.
2. Developing the theory: Identifying, proving or conjecturing fundamental facts.
3. Theorems that adapt the classical "five issues". These theorems are based on results and proof techniques from classical automata theory, but also adapt and reformulate concrete results established more recently in the HS/Automated verification community (Dill, Alur–Henzinger, Maler–Pnueli, Wilke).
4. The lifting of the corresponding heritage raises the following conclusions:
   (a) Lifting to continuous time essentially preserves the heritage for finite signals but the particular choice of oracles has an impact on the questions of determinization, uniformization and decidability.
   (b) $L_2^<$ preserves the status of universal temporal formalism.
   (c) LTS is artificial and subject to ad hoc decisions, especially when expected to specify functions.
   (d) Nets are handled similarly to the original case.
5. But why those five issues, and their adaptation? In what sense are they basic for theory and practice? This is visible already in the original, classical, version and, by analogy, applicable to the extensions.

History and Problems

Since the 1960s the further conceptual development was impressive, as hinted in Table 5.

	Logic	Automata	Nets
Original Trinity (1957–1966)	S1S	String Automata	Logical Nets (Synchronous)
Sugar (Pnueli 1977, Wolper 1983, Kozen 1983, Emerson-Jutla 1991)	Temporal logics μ-calculi	String Automata	Logical Nets (Synchronous)
Branching-time (Rabin 1969, Lamport-Halpern-Emerson)	S2S	Tree Automata	???
Asynchronous interaction (Petri 1962 Kahn 1974)	???	???	Petri Nets, Dataflow networks
Hybrid Systems (Dill, Alur, Henzinger,..., 1990s)	Timed logics	Timed automata, Hybrid Systems	Interaction of control and devices

Table 5: Further conceptual developments

In these lectures, we have confined ourselves to the heritage from the original trinity, ignoring the aspects concerning branching time, asynchrony, etc. The adaptation of the corresponding heritage is an important topic in its own right; hence it is worth looking to other heritages to be lifted, in particular as pertains to the following.

1. Branching time.
2. Asynchronous interaction.
3. Regular expressions.

In the frame of our approach we gathered inspiration and motivation from the vast literature on hybrid systems and timed automata, notably from the following results.

1. Translating monadic formulae into timed automata. (Wilke 1994.)
2. Translating temporal formalisms into monadic logic. (Wilke 1994.)
3. Uniformizing fair automata. (Pnueli, Sifakis, Maler 1995.)
4. Deciding emptiness for timed automata. (Dill 1990.)
5. Various exercises regarding determinization. (Fix, Henzinger, Alur 1995.)

However, in none of these are transducers and nets considered.

Some Problems

1. What are interesting oracle-driven case studies.
2. Reducibilities among oracles.
3. Elaborate on algorithmic and complexity aspects of the five issues. For example, emptiness undecidability may occur even with computable oracles.
4. Bridge to existing theorems, via some friendly sugar.
5. Elaborate the full "uniformization" issue.
6. How to handle branching time?
7. How to extend experiments with automata to continuous time?

Acknowledgements

My views on the subject go back to my past interest and experience in automata theory, as summarized in the books [8, 23], and in the paper [21] (very old recipes). The recent developments in Timed Automata and Hybrid Systems mentioned above stirred up a resurgence of my interest and research in the areas.

This stimulated a research and pedagogical project I started in 1994, at Tel Aviv University. Since then, on different stages, I benefited from the attention and, ultimately, from the collaboration of A. Rabinovich and D. Pardo.

Dorit realized that, if one insists on right-continuous signals, one may need Timers different from those (which are implicit) in current Timed Automata Theory. A final change around this argument, suggested by Alex, led to the full consideration of all non-Zeno signals. Alex worked out the properties of operators even beyond the realm of non-Zeno signals, and managed in this way to prove the Finite-Memory Theorem.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time REX Workshop on Real-Time: Theory in Practice. 1991.

[2] S. Abramsky. Private communication.

[3] R. Alur and D. Dill. Automata for modelling real-time systems. Proceedings of ICALP'90. Lecture Notes in Computer Science 443, pp322–335, 1990.

[4] R. Alur and D. Dill. A theory of timed automata. Theoretical Computer Science, 126:183–235, 1994.

[5] R. Alur and T. Henzinger. Logics and models of real-time: A survey. Proceedings of Real-Time: Theory in Practice. *Lecture Notes in Computer Science*, 600:74–106, 1992.

[6] A. Church. Logic, Arithmetic and Automata. Proceedings of the International mathematical congress, Stockholm, 1962.

[7] S. Kleene. Representation of events in nerve nets and finite automata. Automata Studies, 3-41, Princeton, 1956.

[8] N.E. Kobrinski and B.A. Trakhtenbrot. *Introduction the the theory of finite automata.* North–Holland, Amsterdam, 1965.

[9] L. Lamport and N. Lynch. Distributed computing: models and methods. In Handbook of Theoretical Computer Science, Volume B. Elsevier Science Publishers B.V., 1990.

[10] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer, 1992.

[11] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. *Lecture Notes in Computer Science*, 987:189–???, 1995.

[12] R. McCulloch and W. Pitts. A logical calculus of the ideas immantent in nervous activity. Bull Math. Biophysics, 5, 1943, pp. 115-133.

[13] R. McNaughton. Badly Timed Elements and Well Timed Nets. Technical Report 65-02, Moore School, 1964.

[14] R. Milner. Operational and algebraic semantics of concurrent processes. In Handbook of Theoretical Computer Science, Volume B. Elsevier Science Publishers B.V., 1990.

[15] D. Pardo. Timed Automata: Transducers and Circuits. M.Sc. Thesis, Tel Aviv University, 1997.

[16] D. Pardo, A. Rabinovich and B.A. Trakhtenbrot. On synchronous circuits over continuous time. Technical Report, Tel Aviv Unviersity, 1997.

[17] A. Rabinovich. Finite automata over continuous time. Technical Report, Tel Aviv University, 1996.

[18] A. Rabinovich and B.A. Trakhtenbrot. From Finite Automata toward Hybrid Systems. FCT, 1997.

[19] D. Scott. Some definitional suggestions for automata theory. *Journal of Computer and System Science*, 1967, pp. 187-212.

[20] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, Elsevier, 1990.

[21] B.A. Trakhtenbrot. Finite automata and the monadic predicate calculus. *Sibirsk. Mat. Zh.* (1962), **3**, No. 1, pp 103–131.

[22] B.A. Trakhtenbrot. Origins and Metamorphoses of the Trinity: Logics, Nets, Automata. In *Proceedings of LICS*, 1995.

[23] B.A. Trakhtenbrot and Yu. Barzdin. *Finite Automata: Behaviour and Syntheses.* North-Holland, 1973.

[24] T. Wilke. Specifying Timed State Sequences in Powerful Decidable Logics and Timed Automata. FTRTFT, 1994.