Publications du **La**boratoire de **C**ombinatoire et d' **I**nformatique **M**athématique

15

Edité par S. Brlek

BMW-94 16-18 mai 1994

Méthodes mathématiques pour la synthèse des systèmes informatiques

ACFAS 94

62e Congrès du 16-20 mai 1994
de l'Association Canadienne-Française pour l'Avancement des Sciences.
UQAM, Montréal , Québec , Canada.
Actes du Colloque.

Département de mathématiques et d'informatique

Centre de recherche informatique de Montréal

Université du Québec à Montréal

Edité par:

Srecko Brlek
LACIM
Université du Québec à Montréal
C.P. 8888, Succ. Centre Ville
Montréal, Qc.
Canada H3C 3P8 .

Le colloque sur le thème " Méthodes mathématiques pour la synthèse des systèmes informatiques" a été tenu à Montréal du 16 au 18 mai 1994 dans le cadre du 62e Congrès de l'Association Canadienne-Française pour l'Avancement des Sciences.

Les organisateurs ont bénéficié du support généreux des organismes suivants:

Acfas
Association canadienne-française
pour l'avancement des sciences

Centre de recherche informatique de Montréal
1801, Av. McGill College #800
Montréal, Québec, H3A 2N4

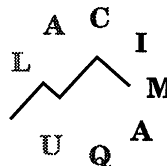
Université du Québec à Montréal

ISBN 2-89276-129-8 LACIM Montréal

© LACIM, Montréal; CRIM, Montréal, Mai 1994.

Le présent numéro a été édité dans le cadre d'une collaboration spéciale entre le CRIM et le LACIM.

Laboratoire de combinatoire et d'informatique mathématique
Département de mathématiques et d'informatique
Université du Québec à Montréal
C.P. 8888, Succ. Centre Ville
Montréal, Qc.
Canada H3C 3P8



BMW-94

Méthodes mathématiques pour la synthèse des systèmes informatiques

Université du Québec à Montréal

16 au 18 mai, 1994

Tenu dans le cadre du congrès de l'ACFAS.
Lieu: Pavillon Judith Jasmin, Université du Québec à Montréal, 405 rue Sainte-Catherine Est (Metro Berri)
Organisé avec le soutien financier de : Action concertée "Recherches Bell-Northern Ltée - Fonds FCAR - CRSNG", PRC-C^3, CRIM, UQAM.

Programme

Lundi 16 mai

9:00–10:15	**Conférence invitée** : K. RUDIE (Queens) *Decentralized Control of Discrete-Event Systems*
10:15–10:30	**Pause café**
10:30–11:15	A. BERGERON (LaCIM) *Easy Problems in Partial Observation*
11:15–12:00	A. KHOUMSI, G.V. BOCHMANN, R. DSSOULI (UdeM) *On Controlling Distributed Communicating Systems*
12:00–13:30	**Dîner libre**
13:30–14:45	**Conférence invitée** : P. RAMADGE (Princeton) *Hybrid Dynamics: Continuous and Discrete Systems*
14:45–15:30	J. THISTLE, R. MALHAMÉ, (École Polytechnique) *Control of discrete-event systems under state fairness assumptions*
15:30–15:45	**Pause café**
15:45–16:30	J.-M. PALMIER, M. MAKUNGU, F.E. AGAPI, M. BARBEAU, R. ST-DENIS (Sherbrooke) *An Introduction to a Synchronized Petri Net Based Tool for the Synthesis of Supervisors of Discrete Event Systems*
16:30–17:15	Y.J. WEI, P.E. CAINES (McGill) *Hierarchical COCOLOG for Finite Machines*
17:15–18:00	B. CHAIB-DRAA, J. DESHARNAIS, R. KHÉDRI, I. JARRAS, S. SAYADI, F. TCHIER (Laval) *Une approche relationnelle à la décomposition parallèle*

Mardi 17 mai

9:00–10:15	**Conférence invitée** : J. F. Groote (Utrecht) *Computer verified, algebraic style verifications of larger protocols*
10:30–11:15	S. Brlek (LaCIM, UQAM), A. Rauzy (CNRS, LaBRI) *Implementation of Constrained Transition Systems: a Unified Approach*
11:15–12:00	A. Griffault (LaBRI, Bordeaux I) *Deux nouvelles techniques d'utilisation de MEC*
12:00–13:30	**Dîner libre**
13:30–14:45	**Conférence invitée** : R. Groz (CNET, France) *Vérification de protocoles: le point de vue d'un opérateur de télécommunications*
14:45–15:30	F. Faci, L. Logrippo (Ottawa) *Specifying Features and Analysing their Interactions in a LOTOS Environment*
15:30–15:45	**Pause café**
15:45–16:30	S. Brlek, R. Mallette (LaCIM, UQAM) *Plain Old Telephone System Service (POTS): Validation avec MEC*
16:30–17:15	J.-C. Grégoire (INRS, Telecom) *The Validation of Buffer-Based Systems*
17:15–18:00	B. Berkane, E. Cerny (UdeM) *Vérification des chronogrammes hiérarchiques à l'aide de "CCS + contraintes"*
19:00–	**Banquet**

Mercredi 18 mai **Activités ayant lieu au Pavillon Carré-Phillips**

–	1193, Place Phillips, 8eme etage.
–	Le programme final de ces activités sera disponible le lundi 16 mai
10:00–	**Démonstrations d'outils**
11:00–	**Discussions informelles**

4

Decentralized Control of Discrete-Event Systems

Karen Rudie
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario K7L 3N6
Canada

Abstract

A summary is given of some of the work on decentralized discrete-event control problems set in the framework initiated by Ramadge and Wonham. A distinction is made between those distributed control problems requiring local specifications and those permitting global specifications. Summarized results include the conditions under which solutions to the problems exist and associated computational complexity bounds.

1 Introduction

Discrete-Event Systems Control, as initiated in the early 1980's by P.J. Ramadge and W.M. Wonham, is the study of discrete-event processes, such as computer systems and manufacturing systems, that require some form of control to achieve desirable behaviour. A discrete-event system is a process or collection of processes that starts out in some initial state and is transformed from state to state by the occurrence of events. Such a system can be thought of as a set of sequences of events, where each sequence describes one possible series of actions that can occur within the system.

A variety of models have been used to represent the behaviour of discrete-event systems. These include, but are not limited to, automata and formal languages [RW82], [Ram83], [Lin87], modal logic [TW86], [Ost89], Petri nets [Kro87], [HK90], and process algebra [IV88], [Ina92], [Hey90]. We consider here only the sampling of work built on the automata/language-theoretic work described in [RW82].

The kinds of problems captured by our formulation are as follows. The process requiring control (called the *plant*) is given by an automaton over an alphabet of event labels. The sequences generated by the automaton characterize the physically possible or unconstrained behaviour of the system. Control is imposed by permitting some events (called *controllable* events) of the system to be prevented from occurring. We then imagine that an external agent (called a *supervisor*) observes the sequences of events generated by the plant and, upon the occurrence of certain sequences, issues commands to disable (i.e., turn off or prevent from occurring) some of the controllable events. For example, suppose that the set of events that may occur is $\{\alpha, \beta, \delta\, \gamma\}$ and suppose that the plant could generate the sequences α, $\alpha\delta$, $\alpha\delta\beta$,

and $\alpha\delta\beta\delta$. If the sequence $\alpha\delta\beta\delta$ were considered "bad", then a supervisor for the plant would need to disable δ after the occurrence of $\alpha\delta\beta$. What if δ were not a controllable event? Now, suppose that δ is a controllable event but that the plant can also generate $\alpha\delta\gamma$ and $\alpha\delta\gamma\delta$. Suppose further that $\alpha\delta\gamma\delta$ is a desirable sequence. What if the supervisor could not observe the occurrence of events γ and β? Then, upon observing the sequence $\alpha\delta$, the supervisor would not know whether $\alpha\delta$, $\alpha\delta\beta$ or $\alpha\delta\gamma$ had actually occurred; if $\alpha\delta\beta$ had occurred, then the supervisor would need to disable δ, whereas if $\alpha\delta\gamma$ had occurred, the supervisor would want to enable δ. We can see that a plant becomes potentially more difficult to control if some of the events are *unobservable*. Now, suppose that, as in many distributed computing problems, several agents act on the system, and each can observe only some subset of events and control only some subset of events. This suggests the question "Can the required control objective be met with the given degree of decentralization?"

This summary paper gives a short overview of some typical problem formulations. We focus on *decentralized* control problems and explore the questions "When are the problems solvable?", "Can we produce solutions when they exist?" and "How difficult is it to compute solutions?".

2 Background

We give a brief review of the framework for discrete-event systems control based on the automata-theoretic model of Ramadge and Wonham. For more details on the formalities of supervisory control theory, the reader is referred to [RW82], [Ram83], [RW87], [WR87], [LW88], [Won88], [WR88], [LW90].

Consider a discrete-event process that can be characterized by an automaton

$$G = (Q, \Sigma, \delta, q_0, Q_m)$$

where Σ is a finite alphabet of event labels (and represents the set of all possible events that can occur within the system), Q is a set of states, $q_0 \in Q$ is the initial state, $Q_m \subseteq Q$ is the set of terminal (often called *marker*) states and $\delta : \Sigma \times Q \longrightarrow Q$, the transition function, is a partial function defined at each state in Q for a subset of Σ. When Q is finite, G can be represented by a directed graph whose nodes are the states in Q and whose edges are transitions defined by δ and labeled by elements from Σ. The automaton G describes the behaviour of a discrete-event process if we interpret transitions as event occurrences that take the process from state to state.

Sequences of concatenated symbols from Σ are interpreted as sequences of events, called *strings*. Let Σ^* denote the set of all finite strings over Σ including the null string ε. Then the transition function δ can be extended to $\Sigma^* \times Q \longrightarrow Q$ by defining $\delta(\varepsilon, q) := q$ and for $s \in \Sigma^*, \sigma \in \Sigma$, $\delta(s\sigma, q) := \delta(\sigma, \delta(s, q))$. That is, we now think of δ as indicating to which state (or states) a *sequence* of events will lead. A subset of Σ^* is called a *language*. The behaviour of the uncontrolled process G, which we call a *plant*, is given by two languages. The *closed behaviour* of G, written $L(G)$, is the language defined as

$$L(G) := \{s \mid s \in \Sigma^* \ and \ \delta(s, q_0) \ is \ defined\}$$

and is interpreted to mean the set of all possible event sequences which the plant could generate. The *marked behaviour* of G, written $L_m(G)$, is the language defined as

$$L_m(G) := \{s \mid s \in \Sigma^* \textit{ and } \delta(s, q_0) \in Q_m\}$$

and is intended to distinguish some subset of possible plant behaviour as representing completed tasks.

To impose supervision on the plant, we identify some of its events as *controllable* and the rest as *uncontrollable*, thereby partitioning Σ into the disjoint sets Σ_c, the set of controllable events, and Σ_{uc}, the set of uncontrollable events. Controllable events are those which an external agent may enable (permit to occur) or disable (prevent from occurring) while uncontrollable events are those which cannot be prevented from occurring and are therefore considered to be permanently enabled. The event set Σ is also partitioned into disjoint sets Σ_o and Σ_{uo} of *observable* and *unobservable* events, respectively. Observable events are those which an external agent may observe during the course of tracking the plant. Given any event set Σ, we may associate with it a mapping, called the *canonical projection*, which we interpret as a supervisor's view of the strings in Σ^*. The projection $P : \Sigma^* \longrightarrow \Sigma_o^*$ is defined as follows: $P(\varepsilon) := \varepsilon$ and for $s \in \Sigma^*, \sigma \in \Sigma$, $P(s\sigma) := P(s)P(\sigma)$, i.e., P erases all unobservable events. If the plant generates a string s, then $P(s)$ is the sequence of events that an external agent observes. Given any language K, the notation $P(K)$ stands for the language $\{P(s) \mid s \in K\}$. The *inverse projection* of P is the mapping $P^{-1} : 2^{\Sigma_o^*} \longrightarrow 2^{\Sigma^*}$ defined on sets of strings (or languages) as $P^{-1}(K) = \{t \mid P(t) \in K\}$. A *supervisor* (sometimes called a *controller*) is then an agent which observes subsequences of the sequences of events generated by G and enables or disables any of the controllable events at any point in time throughout its observation. By performing such a manipulation of controllable events, the supervisor ensures that only a subset of $L(G)$ is permitted to occur. Formally, a *supervisor* $\mathcal{S}$ is a pair (T, ψ) where T is an automaton which recognizes a language over the same event set as the plant G, i.e.,

$$T = (X, \Sigma, \xi, x_0, X_m)$$

where X is the set of states, ξ is the transition function, x_0 is the initial state and X_m are the marker states of the supervisor. The mapping $\psi : \Sigma \times X \rightarrow \{\textit{enable}, \textit{disable}\}$, called a *feedback map*, satisfies

$$\psi(\sigma, x) = \textit{enable}, \text{ if } \sigma \in \Sigma_{uc},\ x \in X$$

and

$$\psi(\sigma, x) \in \{\textit{enable}, \textit{disable}\}, \text{ if } \sigma \in \Sigma_c,\ x \in X.$$

The automaton T is constrained so that

$$\sigma \in \Sigma_{uo}, x \in X \Longrightarrow \xi(\sigma, x) = x.$$

The automaton T tracks the behaviour of G. It changes state according to the observable events generated by G and, in turn, at each state x of T, the control rule $\psi(\sigma, x)$ dictates whether σ is to be enabled or disabled at the corresponding state of G.

The set of sequences of events generated while the plant G is under the control of $\mathcal{S} = (T, \psi)$ characterizes the behaviour of the *closed-loop system* and is represented by

an automaton $\mathcal{S}/G$ whose closed behaviour, denoted by $L(\mathcal{S}/G)$, permits a string to be generated if the string is in both G and T and if each event in the string is enabled by ψ. The marked behaviour of the closed-loop system is denoted by $L_m(\mathcal{S}/G)$ and consists of those strings in $L(\mathcal{S}/G)$ that are marked by both G and S. Formally, the automaton $\mathcal{S}/G$ is given by

$$\mathcal{S}/G := (Q \times X, \Sigma, (\delta \times \xi)^{\psi}, (q_0, x_0), Q_m \times X_m)$$

where $(\delta \times \xi)^{\psi} : \Sigma \times Q \times X \longrightarrow Q \times X$ is defined by

$$(\delta \times \xi)^{\psi}(\sigma, q, x) := \begin{cases} (\delta(\sigma, q), \xi(\sigma, x)) & \text{if both } \delta(\sigma, q), \xi(\sigma, x) \text{ are defined} \\ & \text{and } \psi(\sigma, x) = enable \\ undefined & \text{otherwise.} \end{cases}$$

Often it is important to find supervisors that guarantee that the closed-loop system is *nonblocking*, i.e., that every string generated by the closed-loop system can be completed to a marked string in the system. This requirement is expressed as follows: a supervisor $\mathcal{S}$ is *proper for* G if

$$\overline{L_m(\mathcal{S}/G)} = L(\mathcal{S}/G)$$

where $\overline{K}$ denotes the prefix-closure of a language K.

Typically, control problems require finding for a given plant a supervisor (or set of supervisors) such that the closed-loop system satisfies some prescribed desirable behaviour. Representative centralized supervisory control problems can be found in [RW82], where it is assumed that all events are observable, and [LW88], where it is assumed that some events may not be observable. When controllers act on a given plant, we say that the closed-loop behaviour is *synthesized* by the controllers. Then, control problems involve examining under what conditions prescribed behaviours can be synthesized. Two types of problems may be distinguished: those requiring the behaviour of the closed-loop system to precisely equal some specified behaviour, which we call *synthesis problem without tolerance*; and those requiring the behaviour of the closed-loop system to lie in some specified range, which we call *synthesis problems with tolerance.*

A representative synthesis problem with tolerance is introduced in [RW82] and is as follows:

Supervisory Control Problem (SCP) *Given a plant G over event set Σ, subset $\Sigma_c \subseteq \Sigma$, and languages A, E with $A \subseteq E \subseteq L(G)$, construct a supervisor $\mathcal{S}$ for G which controls only the events in Σ_c, such that $A \subseteq L(\mathcal{S}/G) \subseteq E$.*

The language E embodies the system designer's notion of *legal* or desirable behaviour while A specifies the behaviour common to any acceptable solution, i.e., the *minimally adequate* behaviour. That is, any solution must exhibit at least the behaviour described by A and no more than that described by E.

In order to describe the solution to SCP the notion of *controllability* is defined: A language $K \subseteq L(G)$ is a *controllable sublanguage of* $L(G)$ (or just *controllable* where the associated G is understood) if

$$\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K},$$

where for any languages L and M, the notation LM stands for $\{st \mid s \in L \wedge t \in M\}$. If we interpret $L(G)$ as physically possible behaviour and K as legal behaviour, an informal description of controllability is that K is controllable if for any sequence of events s that starts out as a legal sequence ($s \in \overline{K}$), the occurrence of an uncontrollable event ($\sigma \in \Sigma_{uc}$) which is physically possible ($s\sigma \in L(G)$) does not lead the sequence out of the legal range ($s\sigma \in \overline{K}$).

The class of controllable languages contained in a given language

$$\underline{C}(M,G) := \{K \mid K \subseteq M \textit{ and } K \textit{ is controllable w.r.t. } G\}$$

is nonempty and partially ordered by inclusion; it is closed under arbitrary union and therefore contains a (unique) supremal element, called the *supremal controllable sublanguage of M with respect to (w.r.t.) G*, and denoted by $\sup \underline{C}(M,G)$.

The solution to SCP, given in [RW82], can now be expressed: Provided $A \neq \emptyset$, SCP is solvable if and only if $A \subseteq \sup \underline{C}(E,G)$. If $A = \emptyset$, then SCP is solvable if $\sup \underline{C}(E,G) \neq \emptyset$. Moreover, when G is a finite state automaton and A and E are regular languages, a supervisor $\mathcal{S}$ that synthesizes $\sup \underline{C}(E,G)$ exists with $\mathcal{S}$ having the state transition structure of a recognizer for $\sup \underline{C}(E,G)$. Most important, this $\mathcal{S}$ is an optimal solution in the sense that it permits as much behaviour as possible to occur; it is, therefore, said to be *minimally restrictive.* Algorithms for computing such an $\mathcal{S}$ are given in [LW85] and [Rud88].

3 Decentralized Control: Global Versus Local Specification

Now, we consider the situation where the physical requirements of a problem dictate that decentralized control be used. When a supervisor may act on any controllable event in the entire event set, we say that the supervisor is *global*; in contrast, a supervisor which can only control some subset of controllable events is said to be *local.* Similarly, we speak of a specification as being *global* if it is over the entire plant alphabet and *local* if it is over some subset of the plant alphabet. A decentralized solution prescribes the actions that two or more local supervisors may take. In this paper, we consider the case of two local supervisors.

The decentralized control problems presented below require the following definitions. For supervisors $\mathcal{S}_1 = (T_1, \phi)$ and $\mathcal{S}_2 = (T_2, \psi)$ acting on G with $T_1 = (X, \Sigma, \xi, x_0, X_m)$ and $T_2 = (Y, \Sigma, \eta, y_0, Y_m)$, the *conjunction* of $\mathcal{S}_1$ and $\mathcal{S}_2$ is the supervisor

$$\mathcal{S}_1 \wedge \mathcal{S}_2 := (T_1 \times T_2, \phi * \psi)$$

defined by

$$T_1 \times T_2 := (X \times Y, \Sigma, \xi \times \eta, (x_0, y_0), X_m \times Y_m)$$

with $\sigma \in \Sigma,\ x \in X,\ y \in Y \Longrightarrow$

$$(\xi \times \eta)(\sigma, x, y) := \begin{cases} (\xi(\sigma,x), \eta(\sigma,y)) & \text{if both } \xi(\sigma,x) \text{ and } \eta(\sigma,y) \text{ are defined} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$(\phi * \psi)(\sigma, x, y) := \begin{cases} \textit{disable} & \text{if either } \phi(\sigma,x) = \textit{disable} \text{ or } \psi(\sigma,y) = \textit{disable} \\ \textit{enable} & \text{otherwise.} \end{cases}$$

That is, $T_1 \times T_2$ recognizes the intersection of the languages recognized by T_1 and T_2 and $\phi * \psi$ disables an event if and only if either ϕ or ψ disables it. Thus, $\mathcal{S}_1 \wedge \mathcal{S}_2$ models the actions of $\mathcal{S}_1$ and $\mathcal{S}_2$ operating in parallel. It can be shown [WR88] that $L(\mathcal{S}_1 \wedge \mathcal{S}_2/G) = L(\mathcal{S}_1/G) \cap L(\mathcal{S}_2/G)$ and $L_m(\mathcal{S}_1 \wedge \mathcal{S}_2/G) = L_m(\mathcal{S}_1/G) \cap L_m(\mathcal{S}_2/G)$.

Given a local supervisor $\mathcal{S}$ that controls some subset $\Sigma_{loc,c}$ of Σ_c while observing some subset $\Sigma_{loc,o}$ of Σ, $\tilde{\mathcal{S}}$ denotes the supervisor which takes the same control action as $\mathcal{S}$ on $\Sigma_{loc,c}$, enables all events in $\Sigma \setminus \Sigma_{loc,c}$, makes the same transitions as $\mathcal{S}$ on $\Sigma_{loc,o}$ and stays at the same state for events in $\Sigma \setminus \Sigma_{loc,o}$. The supervisor $\tilde{\mathcal{S}}$ is called the *global extension* of $\mathcal{S}$ (since $\tilde{\mathcal{S}}$ acts on all of Σ while $\mathcal{S}$ acts only on a subset of Σ). A global extension serves as a natural construction of a feasible global supervisor out of a given local supervisor.

We consider two main decentralized control problems, one which requires that specifications be posed in terms of the local event sets that each local supervisor will act on and one which permits specification over the plant's entire event set. The first problem, which we call LP (for *Local Problem*), requires local specification for local control. Sufficient conditions for the existence of a solution to LP have been found by Lin and Wonham [LW90] and the problem is there referred to as DSCOP (for *Decentralized Supervisory Control and Observation Problem*). The second problem, which we call GP (for *Global Problem*), permits global specification for local control. In [RW90] a sufficient condition for the existence of a solution to this problem is given. It is shown in [RW92b] that GP, a synthesis problem with tolerance, can be solved by first considering the case of zero tolerance, which we call GPZT (for *Global Problem with Zero Tolerance*). Necessary and sufficient conditions for the existence of a solution to GPZT have been found by Cieslak *et al.* [CDFV88]. Work by Inan [Ina92] yields comparable results. Willner and Heymann [WH90] also examine the GPZT formulation for the special case where the plant G can be represented as a synchronous product of processes.

The distinction between local specifications and global specifications can be summarized as follows. Each local specification describes the task that one supervisor must perform. So, e.g., in a manufacturing system of machines depositing parts in buffers one constraint might be that some buffer not overflow. We'd have to construct a controller whose job is to guarantee that that buffer not overflow. In contrast, a global specification describes a joint task that two or more controllers perform. Consider a communication system: each agent's protocol is a recipe for behaviour of that agent and can be thought of as a controller. Then, a typical constraint might be that "data that has been sent by a Sender must be printed out by a Receiver in the correct order". In this case, our goal would be to construct *two* controllers (i.e., one recipe for the Sender and one for the Receiver) whose jobs are to together ensure that data flow in the prescribed way. There is no *one* controller whose task is to control data.

The following problem formulation, considered in [Lin87] and [LW90], captures problems where specifications are given as a collection of local constraints.

Local Problem (LP) *Given a plant G over an alphabet Σ, legal languages E_1, E_2 over alphabets $\Sigma_1 \subseteq \Sigma$, $\Sigma_2 \subseteq \Sigma$ (resp.), and minimally adequate languages $A_1 \subseteq E_1$, $A_2 \subseteq E_2$, sets $\Sigma_{1,c}, \Sigma_{1,o} \subseteq \Sigma_1$ and $\Sigma_{2,c}, \Sigma_{2,o} \subseteq \Sigma_2$, construct local supervisors $\mathcal{S}_1$ and $\mathcal{S}_2$ such that*

$$L(G) \cap \bigcap_{i=1}^{2} T_i^{-1} A_i \subseteq L(\tilde{\mathcal{S}}_1 \wedge \tilde{\mathcal{S}}_2/G) \subseteq L(G) \cap \bigcap_{i=1}^{2} T_i^{-1} E_i$$

Here, for $i = 1, 2$, T_i is the projection from Σ^* to Σ_i^*, S_i is a supervisor which can observe only events in $\Sigma_{i,o}$ and can control only events in $\Sigma_{i,c}$, and $\tilde{S}_i$ is the global extension of S_i.

The reason we speak of LP as requiring local specifications is that the languages A_1, E_1 and A_2, E_2 describe desirable behaviour in terms of the local event sets Σ_1 and Σ_2, respectively.

Formulations requiring local specification are not suitable for modeling communication problems because specifications for communication networks are typically given as global requirements. That is, a problem statement describes what goal the network as a whole must achieve (e.g., what data is to be transferred to which location) without spelling out what each agent in the network must do to achieve this goal. The protocol for each agent is part of the *solution* to the problem and is better described by the languages $L(\tilde{S}_1/G)$ and $L(\tilde{S}_2/G)$ which accompany the solution to GP, presented below, than by any solution to LP. A communication example modeled by GP is given in [RW90].

Global Problem (GP) *Given a plant G over an alphabet Σ, a legal language $E \subseteq L_m(G)$, a minimally adequate language $A \subseteq E$, and sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, construct local supervisors S_1 and S_2 such that $\tilde{S}_1 \wedge \tilde{S}_2$ is a proper supervisor for G and such that*

$$A \subseteq L(\tilde{S}_1 \wedge \tilde{S}_2/G) \subseteq E$$

Here, for $i = 1, 2$, supervisor S_i can observe only events in $\Sigma_{i,o}$ and control only events in $\Sigma_{i,c}$ and where $\tilde{S}_i$ is the global extension of S_i. The set of uncontrollable events, Σ_{uc} is understood to be $\Sigma \setminus (\Sigma_{1,c} \cup \Sigma_{2,c})$.

The assumption here is that for LP, specifications can be thought of as a conjunction or collection of tasks where the goal is to find several supervisors, each of which accomplishes one of the tasks. In contrast, the specifications of GP cannot necessarily be broken up into subtasks, each of which is performed by only one supervisor.

4 A Summary of Results

4.1 Necessary and Sufficient Conditions

Sufficient conditions under which a solution to LP exists are contained in [LW90]. When such conditions hold, the solution roughly amounts to locally solving centralized control problems for each of the two sets of local specifications. Moreover, in [LW90], a manufacturing system problem is cast as LP and solved, suggesting that LP is an appropriate formulation for at least some distributed systems problems. However, the solution to LP does not guarantee nonblocking even if both $\tilde{S}_1$ and $\tilde{S}_2$ are proper.

To arrive at a sufficient condition for GP to be solvable, we note that GP can be reduced to LP for the class of systems characterized by the following property, first introduced in [RW90]. For a language $K \subseteq L(G)$, K is *decomposable* w.r.t. G and projections P_1, P_2 if

$$K = L(G) \cap P_1^{-1}(P_1(K)) \cap P_2^{-1}(P_2(K))$$

The inclusion $K \subseteq L(G) \cap P_1^{-1}(P_1(K)) \cap P_2^{-1}(P_2(K))$ is automatic (by definition of projection and the fact that $K \subseteq L(G)$); informally then, a language is decomposable only if the plant G and the local versions of K ($P_1(K)$ and $P_2(K)$) contain enough information to permit the global K to be reconstructed. As demonstrated in [RW90], for the case where $\Sigma_{1,o} = \Sigma_1$ and $\Sigma_{2,o} = \Sigma_2$ in the LP formulation, the reduction of GP to LP is possible if the languages A and E are decomposable with respect to G, P_1, P_2, where P_1 and P_2 are the projections onto $\Sigma_{1,o}$ and $\Sigma_{2,o}$, respectively. However, decomposability is not *necessary*. Moreover, it is a strong assumption since it requires that an acceptable controller be able to observe any event that it disables.

It is shown in [RW92b] that a property weaker than decomposability, called *co-observability*, plays a key role in the necessary and sufficient conditions for solving GP. To solve GP, one first considers the special case where A equals E. It can be shown that a solution to the general case can be derived from the solution to the special case. The latter is formulated as follows.

Global Problem with Zero Tolerance (GPZT) *Given a plant G over an alphabet Σ, a legal language E such that $\emptyset \neq E \subseteq L_m(G)$ and sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, construct local supervisors $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\tilde{\mathcal{S}_1} \wedge \tilde{\mathcal{S}_2}$ is a proper supervisor for G and such that*

$$L_m(\tilde{\mathcal{S}}_1 \wedge \tilde{\mathcal{S}}_2/G) = E$$

Here again, for $i = 1, 2$, supervisor $\mathcal{S}_i$ can observe only events in $\Sigma_{i,o}$ and can control only events in $\Sigma_{i,c}$ and $\tilde{\mathcal{S}}_i$ is the global extension of $\mathcal{S}_i$. Again, the set of uncontrollable events is taken to be $\Sigma \setminus (\Sigma_{1,c} \cup \Sigma_{2,c})$.

By assumption, solutions to GP and GPZT are always *nonblocking* since a pair of supervisors $\mathcal{S}_1, \mathcal{S}_2$ is considered a solution to either only if $\tilde{\mathcal{S}}_1 \wedge \tilde{\mathcal{S}}_2$ is a *proper* supervisor for G. In contrast, as stated earlier, solutions to LP are not necessarily nonblocking.

Observe that for the global control problem with zero tolerance, the goal is to find supervisors such that the *marked behaviour* of the closed-loop system is precisely equal to some given language. In contrast, when nonzero tolerance is permitted, it suffices to find supervisors such that the *closed behaviour* of the closed-loop system is contained in some given range of languages. Consequently, it may often be assumed for the nonzero tolerance case, but not for the zero tolerance case, that the languages under consideration are prefix-closed.

It can be seen that LP and GPZT are, in fact, special cases of GP. The LP formulation can be restated as a version of GP where the specifications A and E of GP are given as $L(G) \cap \bigcap_{i=1}^{2} T_i^{-1} A_i$ and $L(G) \cap \bigcap_{i=1}^{2} T_i^{-1} E_i$, respectively. That is, local specifications are simply a structured case of global specifications. Similarly, for prefix-closed specifications, GPZT fits into the GP formulation if we let the endpoints of the range of behaviour in GP be equal, i.e., if $A = E$.

The solution to both GPZT and GP rely on a property called *co-observability*, defined in [RW92b] as follows. Given a plant G over alphabet Σ, sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, projections $P_1 : \Sigma^* \longrightarrow \Sigma_{1,o}^*$, $P_2 : \Sigma^* \longrightarrow \Sigma_{2,o}^*$, a language $K \subseteq L_m(G)$ is *co-observable w.r.t.* G, P_1, P_2 if

$s, s', s'' \in \Sigma^*,\ P_1(s) = P_1(s'),\ P_2(s) = P_2(s'') \Longrightarrow$

$$
\begin{array}{lll}
 & (\forall \sigma \in \Sigma_{1,c} \cap \Sigma_{2,c})\ s \in \overline{K} \wedge s\sigma \in L(G) \wedge s'\sigma, s''\sigma \in \overline{K} \Longrightarrow s\sigma \in \overline{K} & \textbf{conjunct 1} \\
\wedge & (\forall \sigma \in \Sigma_{1,c} \setminus \Sigma_{2,c})\ s \in \overline{K} \wedge s\sigma \in L(G) \wedge s'\sigma \in \overline{K} \Longrightarrow s\sigma \in \overline{K} & \textbf{conjunct 2} \\
\wedge & (\forall \sigma \in \Sigma_{2,c} \setminus \Sigma_{1,c})\ s \in \overline{K} \wedge s\sigma \in L(G) \wedge s''\sigma \in \overline{K} \Longrightarrow s\sigma \in \overline{K} & \textbf{conjunct 3} \\
\wedge & s \in \overline{K} \cap L_m(G) \wedge s', s'' \in K \Longrightarrow s \in K. & \textbf{conjunct 4}
\end{array}
$$

Intuitively, a supervisor knows what action to take if it knows what sequence of events actually occurred. However, a string which, for each supervisor, looks like (i.e., has the same projection as) another string may be potentially ambiguous in determining control action. On this basis, if we assume that some external agent, such as a supervisor, determines which strings are allowed to be in $\overline{K}$ and which in K, an informal description of co-observability is as follows. A language K is co-observable if (1) after the occurrence of an ambiguous string, s, in $\overline{K}$, the decision to enable or disable a controllable event σ is forced by the action that a supervisor which can control σ would take on other strings which look like s (encompassed by conjuncts (1)–(3) in the definition of co-observability), and (2) the decision to mark or not mark a potentially confusing string is determined by at least one of the supervisors (covered by conjunct (4)). Note that if a language K is prefix-closed, then conjunct (4) always holds.

It was shown in [CDFV88] and [RW92b] that GPZT is solvable iff the legal language E is controllable and co-observable w.r.t. G. In [RW92a], co-observability is used to check for safety properties in a communication protocol.

We return to the case when the desired behaviour is given by a range of languages $[A, E]$, where A does not necessarily equal E. Since we know from the above result that the only languages within the range $[A, E]$ that can be synthesized are those that are controllable and co-observable, to solve the more general problem GP, we must find a controllable and co-observable language containing A and contained in E. It is shown in [RW92b] that the infimal, prefix-closed, controllable and co-observable language containing a given language can be computed. Therefore, GP is solvable iff and the infimal prefix-closed, controllable and co-observable language containing A is contained in E. That is, we take the lower bound on desired behaviour, A, and add to it event sequences according to the algorithm given in [RW92b] just until we get a controllable and co-observable language and we check if the language we now have is still contained in E. If so, we find decentralized controllers that synthesize that language (which we can do in the manner prescribed by the algorithm for finding controllers that solve GPZT).

4.2 Computational Complexity

All computational complexity results cited in this section assume that the plant G and specification languages A and E are given as finite-state machines.

Centralized discrete-event control problems of the type described in this paper essentially rely on computing controllability. It is shown in [WR88] that controllability of E w.r.t. G can be decided in polynomial time w.r.t. the number of states in G and E. Furthermore, the supremal controllable sublanguage of a given language can also be computed in polynomial time. Since, as stated above, the solution of LP, the control problem with local specifications, reduces to solving two centralized control problems, we can check whether LP is solvable in polynomial time and, if it is, we can compute supervisors in polynomial time.

The two global-specification problems pose a bigger problem. While it was shown in [RW93] that co-observability can be decided in polynomial time, the negative complexity results of centralized control problems with partial observability given in [Tsi89] can be generalized to show that the infimal prefix-closed, controllable and co-observable language containing a given language *cannot* be computed in polynomial time. This means that, while one can check in polynomial time whether GPZT is solvable, the same is not true for GP. Moreover, even if a language E *is* controllable and co-observable, Tsitsiklis' results indicate that decentralized supervisors that synthesize E cannot be constructed in polynomial time [Tsi89].

5 Conclusions

We have reviewed some of the work in decentralized discrete-event control theory, making the distinction between those problems requiring local specifications and those admitting global specifications. While the global-specification formulation is more general, and, we believe, more appropriate for communication problems, it is not always computationally efficient to produce solutions to problems. On the other hand, since checking the property of co-observability can be performed in polynomial-time and since we believe that some communication protocol problems can be reformulated as discrete-event control problems where co-observability is the main test for solvability, it may be practical to use supervisory control theory to perform protocol verification. We believe that reworking problems into a control-theoretic framework could prove helpful in analyzing and systematically solving distributed computer systems problems.

References

[CDFV88] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, March 1988.

[Hey90] M. Heymann. Concurrency and discrete event control. *IEEE Control Systems Magazine*, pages 103–112, June 1990.

[HK90] L. E. Holloway and B. H. Krogh. Synthesis of feedback control logic for a class of controlled Petri nets. *IEEE Transactions on Automatic Control*, 35(5):514–523, May 1990.

[Ina92] K. Inan. An algebraic approach to supervisory control. *Mathematics of Control, Signals, and Systems*, 5(2):151–164, 1992.

[IV88] K. Inan and P. Varaiya. Finitely recursive process models for discrete event systems. *IEEE Transactions on Automatic Control*, 33(7):626–639, July 1988.

[Kro87] B. H. Krogh. Controlled Petri nets and maximally permissive feedback logic. In *Proceedings of the 25th Annual Allerton Conference on Communication, Control and Computing*, pages 317–326, University of Illinois, Urbana, 1987.

[Lin87] F. Lin. *On Controllability and Observability of Discrete Event Systems.* PhD thesis, Department of Electrical Engineering, University of Toronto, 1987.

[LW85] F. Lin and W. M. Wonham. On the computation of supremal controllable sublanguages. In *Proceedings of the 23rd Annual Allerton Conference on Communication, Control and Computing*, pages 942–950, University of Illinois, Urbana, 1985.

[LW88] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44:173–198, 1988.

[LW90] F. Lin and W. M. Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions on Automatic Control*, 35(12):1330–1337, December 1990.

[Ost89] J. S. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studies Press, Somerset, England, 1989.

[Ram83] P. J. Ramadge. *Control and Supervision of Discrete Event Processes.* PhD thesis, Department of Electrical Engineering, University of Toronto, 1983.

[RW82] P. J. Ramadge and W. M. Wonham. Supervision of discrete event processes. In *Proceedings of the 21st IEEE Conference on Decision and Control*, volume 3, pages 1228–1229, December 1982.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[Rud88] K. G. Rudie. Software for the control of discrete event systems: A complexity study. Master's thesis, Department of Electrical Engineering, University of Toronto, 1988.

[RW93] K. Rudie and J. C. Willems. The computational complexity of decentralized discrete-event control problems. In *Proceedings of the European Control Conference*, pages 2185–2190, Groningen, The Netherlands, June 28 – July 1 1993.

[RW90] K. Rudie and W. M. Wonham. Supervisory control of communicating processes. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification X*, pages 243–257. Elsevier Science (North-Holland), 1990.

[RW92a] K. Rudie and W. M. Wonham. Protocol verification using discrete-event systems. In *Proceedings of the 31st IEEE Conference on Decision and Control*, pages 3770–3777, Tucson, Arizona, December 1992.

[RW92b] K. Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, November 1992.

[TW86] J. G. Thistle and W. M. Wonham. Control problems in a temporal logic framework. *International Journal of Control*, 44(4):943–976, 1986.

[Tsi89] J. N. Tsitsiklis. On the control of discrete-event dynamical systems. *Mathematics of Control, Signals, and Systems*, 2:95–107, 1989.

[WH90] Y. Willner and M. Heymann. On supervisory control of concurrent discrete-event systems. Computer Science Department CIS Report #9009, Technion—Israel Institute of Technology, 1990.

[Won88] W. M. Wonham. A control theory for discrete-event systems. In M. J. Denham and A. J. Laub, editors, *Advanced Computing Concepts and Techniques in Control Engineering*, volume F47 of *NATO ASI Series*, pages 129–169. Springer-Verlag, Berlin, 1988.

[WR87] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, 1987.

[WR88] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems*, 1:13–30, 1988.

Easy Problems in Partial Observation

Anne Bergeron*
LACIM, Université du Québec à Montréal
C. P. 8888 Succ. Centre-Ville
Montréal, Canada, H3C 3P8
e-mail: anne@lacim.uqam.ca

Introduction

This paper discusses computational issues related to observation of distributed discrete processes. This is a central problem in the theory of control of discrete-event processes initiated by Ramadge and Wonham [Ramadge & Wonham, 89; Rudie & Wonham, 92]. Given a process **P** with k possible states, an *observer* is another process **O** that monitors **P** in order to, for example, report completed tasks, restrict the possible behaviors, or diagnose faults. In the absence of complete information about occurrence of events in **P**, it is generally impossible to determine accurately the current state of **P**. Subsets of states being undistinguishable, the number of states of an observer can be exponential in k, the number of states of **P** [Tsitsiklis, 89]. This situation gets worse in distributed processes, where we usually want to construct several observers [Rudie & Willems, 93].

We study here some classes of processes where is can be shown that, far from being exponential, the number of states of an observer is *less* than the number of states of the process being monitored. Such *linearly observable processes* form the basis of what we hope to be a computationally feasible approach to control problems in distributed processes. This approach is of practical importance since various formalism used to describe distributed processes are known to lead to some kind of state explosion. Software tools have been developed to analyze those specification, and they can handle gracefully automata that have several thousands of states [Arnold, 89; 90]. With specifications of this size, however, the prospect of constructing controllers using exponential algorithms is completely unrealistic.

1. Automata and Discrete Processes

Let Σ be a finite set whose elements are called *events*, and Σ^* bet the set of all finite sequences of elements of Σ. An *automaton* **A** on the set Σ of events is given by an arbitrary *partial* function – multiplicatively denoted by a dot "$\cdot$" – called a *transition function*:

$$\cdot : S_A \times \Sigma \longrightarrow S_A$$

where S_A is an arbitrary set, called the *states* of **A**. Every transition function can be naturally extended to any sequence x in Σ^* in the following way:

i) $s \cdot \lambda = s$ where λ is the empty sequence,
ii) $s \cdot (x\sigma) = (s \cdot x) \cdot \sigma$ whenever the right hand side is defined.

* This work was partially supported by grants from BNR Ltd., FCAR of Québec and NSERC of Canada.

Among the states S_A, we distinguish an *initial state* i_A and a subset $F_A \subseteq S_A$ of *final* or *marked states*. We will denote by $\overline{A}$ the automaton obtained by marking all the states of an automaton **A**. The language *recognized* by the automaton **A** is the set $L(A) = \{ x \mid i \cdot x \in F_A \}$. A state s is *accessible* if there is at least one sequence x such that $i_A \cdot x = s$. We will always assume that S_A contains only accessible states.

Definition 1.1 *Product of Automata*

Given two automata **A** and **B**, the *product* $A \times B$ has states $S = S_A \times S_B$ and transition function:

$$\cdot : S \times \Sigma^* \longrightarrow S$$

where $(s,t) \cdot x = (s \cdot x, t \cdot x)$ whenever the right hand side is defined.

The product has initial state $i = (i_A, i_B)$ and marked states $F = \{ (s, t) \mid s \in F_A \text{ and } t \in F_B \}$.

■

In order to compare the behavior of automata or product of automata, we define two partial orders on automata. The first one is based on the languages recognized by automata, and the second one, much stronger, is based on the existence of a morphism between their algebraic structure:

Definition 1.2 *Morphisms between Automata*

Let **A**, **B** be automata with states S_A, S_B initial states i_A, i_B, and final states F_A, F_B. A *morphism* from **A** to **B** is defined by a function $f : S_A \longrightarrow S_B$ such that $f(i_A) = i_B$, $f(F_A) \subseteq F_B$, and if $s \cdot \sigma$ is defined then $f(s) \cdot \sigma$ is also defined and equal to $f(s \cdot \sigma)$, that is, the following diagram commutes:

$$\begin{array}{ccc} S_A \times \Sigma & \xrightarrow{<f, 1_\Sigma>} & S_B \times \Sigma \\ \downarrow & & \downarrow \\ S_A & \xrightarrow{f} & S_B \end{array}$$

■

Definition 1.3 *The relation $A \leq B$*

Let **A** and **B** be two automata. The relation $A \leq B$ holds whenever we have $L(A) \subseteq L(B)$ and $L(\overline{A}) \subseteq L(\overline{B})$. When both $A \leq B$ and $B \leq A$, we write $A \approx B$.

■

Definition 1.3 (bis) *The relation $A \longrightarrow B$*

Let **A** and **B** be two automata. The relation $A \longrightarrow B$ holds whenever there is a morphism from **A** to **B**. When both $A \longrightarrow B$ and $B \longrightarrow A$, then **A** and **B** are said to be *isomorphic*, and we will write $A \longleftrightarrow B$.

■

If $\mathbf{A} \longrightarrow \mathbf{B}$ then the function $f: S_A \longrightarrow S_B$ is unique. Occasionally, we will refer explicitly to this function with the notation $\mathbf{A} \xrightarrow{f} \mathbf{B}$. If $\mathbf{A} \xrightarrow{f} \mathbf{B}$ and $\mathbf{B} \xrightarrow{g} \mathbf{C}$ then $\mathbf{A} \longrightarrow \mathbf{C}$, by composition of g and f. If $\mathbf{A} \xrightarrow{f} \mathbf{B}$, and if the function f is injective, $\mathbf{A}$ is a *sub-automaton* of $\mathbf{B}$, and we will use the notation $\mathbf{A} \hookrightarrow \mathbf{B}$.

Elementary properties of these definitions will be used throughout this paper:

Proposition 1.4 Let $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ be any automata:

i) $\mathbf{A}\times\mathbf{B} \leq \mathbf{A}$ and $\mathbf{A}\times\mathbf{B} \leq \mathbf{B}$.

ii) $\mathbf{A} \leq \mathbf{B}$ if and only if $\mathbf{A} \approx \mathbf{B} \times \mathbf{A}$.

iii) $\mathbf{C} \leq \mathbf{A}\times\mathbf{B}$ if and only if $\mathbf{C} \leq \mathbf{A}$ and $\mathbf{C} \leq \mathbf{B}$.

■

Proposition 1.5 Let $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ be any automata, then:

i) $\mathbf{A}\times\mathbf{B} \longrightarrow \mathbf{A}$ and $\mathbf{A}\times\mathbf{B} \longrightarrow \mathbf{B}$

ii) $\mathbf{A} \longrightarrow \mathbf{B}$ if and only if $\mathbf{A} \longleftrightarrow \mathbf{A} \times \mathbf{B}$.

iii) $\mathbf{C} \longrightarrow \mathbf{A}\times\mathbf{B}$ if and only if $\mathbf{C} \longrightarrow \mathbf{A}$ and $\mathbf{C} \longrightarrow \mathbf{B}$.

■

Finally, we have that:

Proposition 1.6 If $\mathbf{A} \longrightarrow \mathbf{B}$ then $\mathbf{A} \leq \mathbf{B}$.

■

If $\mathbf{A} \xrightarrow{f} \mathbf{B}$, we will note $f(\mathbf{A})$ the *image* automaton of $\mathbf{A}$. That is, the states of $f(\mathbf{A})$ are:

$$S_{f(\mathbf{A})} = \{f(s) \mid s \in S_{\mathbf{A}}\}$$

with initial state $f(\mathbf{i_A})$, final states $f(\mathbf{F_A})$, and transition function:

$$\cdot : S_{f(\mathbf{A})} \times \Sigma \longrightarrow S_{f(\mathbf{A})}$$

where $\mathbf{t}\cdot\sigma$ is defined iff $\mathbf{s}\cdot\sigma$ is defined in $\mathbf{A}$ for at least one state $\mathbf{s}$ in $f^{-1}(\mathbf{t})$. Note that if $\mathbf{A}$ has k states, then $f(\mathbf{A})$ has at most k states. We have the following decomposition:

Proposition 1.7 If $\mathbf{A} \xrightarrow{f} \mathbf{B}$, then $\mathbf{A} \longrightarrow f(\mathbf{A}) \hookrightarrow \mathbf{B}$.

■

2. Observation and Linear Observation

Let **P** be a process, a *specification* for the process **P** will be modeled by an automaton $\mathbf{Z} \leq \mathbf{P}$. The specification can be tought of as the set of acceptable behaviors of the process **P**. Suppose we are given n sites, each of them having a set of unobservable events Σ_{u_i}. Our objective is to construct n automata, one at each site, such that when these automata function simultaneously with the process **P**, the global process 'meets' all specified behaviors, and all illegal behaviors are 'prevented'. This somewhat vague statement can be formalized within automata theory in the following way. Suppose that $\mathbf{C}_1, ..., \mathbf{C}_n$ are n automata functioning with the process **P**. Consider the product

$$\mathbf{P} \times \Pi \mathbf{C}_i.$$

If $\mathbf{C}_i$ must function with only the information available at site i, it cannot change state upon the occurrence of events in the set Σ_{u_i} of unobservable events at site i. Such an automata is called and observation automata, or an observer, with respect to Σ_{u_i}:

Definition 2.1 *Observation Automata*

An automaton **O** is an *observation automaton* with respect to Σ_u, its *unobservable events*, if whenever $s \cdot \sigma$ is defined and $\sigma \in \Sigma_u$ we have $s \cdot \sigma = s$.

■

If the global process is to meet all specified behaviors, we must have:

$$\mathbf{Z} \leq \mathbf{P} \times \Pi \mathbf{C}_i.$$

That is, any sequence defined or marked by **Z**, must also be defined and marked by $\mathbf{P} \times \Pi \mathbf{C}_i$.

On the other hand, the purpose of the automata $\Pi \mathbf{C}_i$ is to restrict the behavior of **P**, such that only sequences defined or marked by **Z** are defined. Thus we want also that:

$$\mathbf{P} \times \Pi \mathbf{C}_i \leq \mathbf{Z}$$

The next definition sums up these conditions:

Definition 2.2 *Observable Specifications*

Given n sites with unobservable events Σ_{u_i}, a specification **Z** of a process **P** is *observable* if there exists automata $\mathbf{C}_1, ..., \mathbf{C}_n$ such that:

i) $\mathbf{C}_i$ is an observation automata with respect to Σ_{u_i},
ii) $\mathbf{Z} \approx \mathbf{P} \times \Pi \mathbf{C}_i$

■

Given any automaton A, and a set of unobservable events Σ_u, it is always possible to construct an observation automaton greater than A, and which is minimal among all observation automata greater than A. This construction is basic in partial observation problems and is a variant of the well-known determinization algorithm in automata theory:

Algorithm 2.3 *The Minimal Observer Construction*

Let $\mathbf{S_A}$ be the set of states of an automaton A, with initial state **i** and marked states $\mathbf{F_A}$. Let Σ_u be a set of unobservable events. The *minimal observer* $\mathbb{O}$(A) with respect to Σ_u is defined in the following way. The states of the automaton $\mathbb{O}$(A) are non-empty subsets $\mathcal{P}^+(\mathbf{S_A})$ of $\mathbf{S_A}$. The initial state is:

$$\mathbf{I} = \{ \mathbf{i}\cdot u \mid \mathbf{i}\cdot u \text{ is defined and } u \in \Sigma_u^* \}$$

and the marked states of $\mathbb{O}$(A) are all subsets that contain at least one marked state of A. The transition function

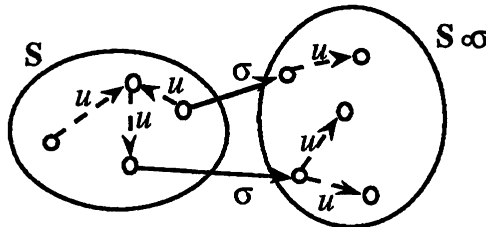
$$\circ: \mathcal{P}^+(\mathbf{S_A}) \times \Sigma \longrightarrow \mathcal{P}^+(\mathbf{S_A})$$

is defined with the following rule:

Let $\mathbf{S} \subseteq \mathbf{S_A}$, if $\mathbf{s}\cdot\sigma$ is defined in A for at least one $\mathbf{s} \in \mathbf{S}$ then $\mathbf{S}\circ\sigma$ is defined and
i) If $\sigma \notin \Sigma_u$, $\mathbf{S}\circ\sigma = \{\mathbf{s}\cdot\sigma u \mid \mathbf{s} \in \mathbf{S}, \mathbf{s}\cdot\sigma u \text{ is defined and } u \in \Sigma_u^*\}$
ii) If $\sigma \in \Sigma_u$, $\mathbf{S}\circ\sigma = \mathbf{S}$
otherwise $\mathbf{S}\cdot\sigma$ is undefined.

■

Informally, the initial state of $\mathbb{O}$(A) is the set of states reachable from the initial state $\mathbf{i_A}$ with unobservable sequences: the observer cannot distinguish between these states. The construction then proceeds recursively from the initial state. Suppose the observer 'thinks' it could be in any of the states of a subset S, if an observable event σ occurs, we have to compute all the states reachable with unobservable sequences from states of the form s·σ, where s is in S.



The following result summarizes the basic properties of the minimal observer construction. Its proof can be found in [Bergeron, 93].

Theorem 2.4 Let **A** be any automata, and Σ_u be a set of events, then:

i) $\mathbb{O}$(**A**) is an observation automata with respect to Σ_u.

ii) **A** $\leq \mathbb{O}$(**A**).

iii) If **X** is an observation automata wrt Σ_u such that **A** $\leq$ **X**, then $\mathbb{O}$(**A**) $\leq$ **X**.

■

The next theorem gives a general criterion for observability. It says that, in order to establish observability, it suffices to consider the minimal observers of **Z**.

Theorem 2.5 Given n sites with unobservable events Σ_{ui}. Let **Z** be a specification of a process **P**, and let $\mathbb{O}_i$(**Z**) be the minimal observer with respect to Σ_{ui}, then

Z is observable if and only if $\mathbf{Z} \approx \mathbf{P} \times \prod \mathbb{O}_i(\mathbf{Z})$.

■

Algorithm 2.3 can – and does – lead to computational disaster since the number of states of an observer can be exponential in the number of states of the specification [Tsitsiklis, 89; Rudie & Willems, 93]. The purpose of the next definition is to restrict the possible observers of **Z**:

Definition 2.4 *Linearly Observable Specifications*

Given n sites with unobservable events Σ_{ui}, a specification **Z** of a process **P** is *linearly observable* if there exists automata $\mathbf{C}_1, ..., \mathbf{C}_n$ such that:

i) $\mathbf{C}_i$ is an observation automata with respect to Σ_{ui},

ii) $\mathbf{Z} \longleftrightarrow \mathbf{P} \times \prod \mathbf{C}_i$

■

Clearly, a linearly observable specification is observable. The terminology *linear* comes from the fact that we can bound the number of states of the various observers by the number of states of the specification.

Proposition 2.5 If **Z** is linearly observable, and if **Z** has k states, then it is linearly observable by automata that have at most k states.

■

Proposition 2.5 tells us that, in the presence of linear observability, there is no state explosion of the observers $\mathbf{C}_i$, and the number of states of the product $\mathbf{P} \times \prod \mathbf{C}_i$ is equal to the number of states of the specification **Z**, thus ensuring efficient computations. In the sequel, we will want to identify conditions ensuring linear observability, or, even better, classes of processes where it can be proved that observability is equivalent to linear observability.

If $\mathbf{Z}$ is linearly observable by $\Pi \mathbf{C}_i$, then $\mathbf{Z} \longrightarrow \mathbf{C}_i$. When $\mathbf{Z}$ is a sub-automaton of $\mathbf{P}$, and if $\mathbf{Z}$ is observable $\Pi \mathbf{C}_i$, these morphism are sufficient to infer linear observability:

Theorem 2.6 Let $\mathbf{Z} \hookrightarrow \mathbf{P}$ be a specification of a process $\mathbf{P}$, if $\mathbf{Z}$ is observable by $\Pi \mathbf{C}_i$, and if, for each i, $\mathbf{Z} \longrightarrow \mathbf{C}_i$, then $\mathbf{Z}$ is linearly observable.

■

3. Exchange Networks

Exchange networks share features with Petri nets and vector addition systems [Yong & Wonham, 93]. Additional structure on the states of automata arising in this context will provide elegant and efficient ways of obtaining observers $\mathbf{C}_i$, and morphisms $\mathbf{Z} \longrightarrow \mathbf{C}_i$, which are a necessary condition for linear observability.

Definition 3.1 *Exchange networks*

An *exchange network* $\mathbf{E}$ is a graph whose nodes $\mathbb{P} = \{p_1, ..., p_k\}$ are called *places*, and vertices $\mathbb{C}$ are called *channels*. The source and target of each channel is given by functions:

$$s, t\colon \mathbb{C} \longrightarrow \mathbb{P}$$

A *configuration* is a function $C\colon \mathbb{P} \longrightarrow \mathbb{N}$ which assign to each place p the natural number $C(p)$ of *tokens* in the place.

■

We will be interested in the various configurations obtained by moving tokens along the channels. An elementary *move* d will correspond to the transfer of a token from a place p_i to a place p_j, if

1) there is a channel d such that $s(d) = p_i$ and $t(d) = p_j$;
2) the place p_i is not empty, that is $C(p_i) \neq 0$.

A configuration C_1 is *accessible* from a configuration C_2 if there is a sequence of elementary moves transforming C_2 into C_1. Consider the set of all possible configurations with N tokens in a network with k places, we call this set the *simplex* $\Delta_{N,k}$. This is the set points $(n_1, ..., n_k)$ of $\mathbb{N}^k$ defined by the equation the equation $\sum n_i = N$.

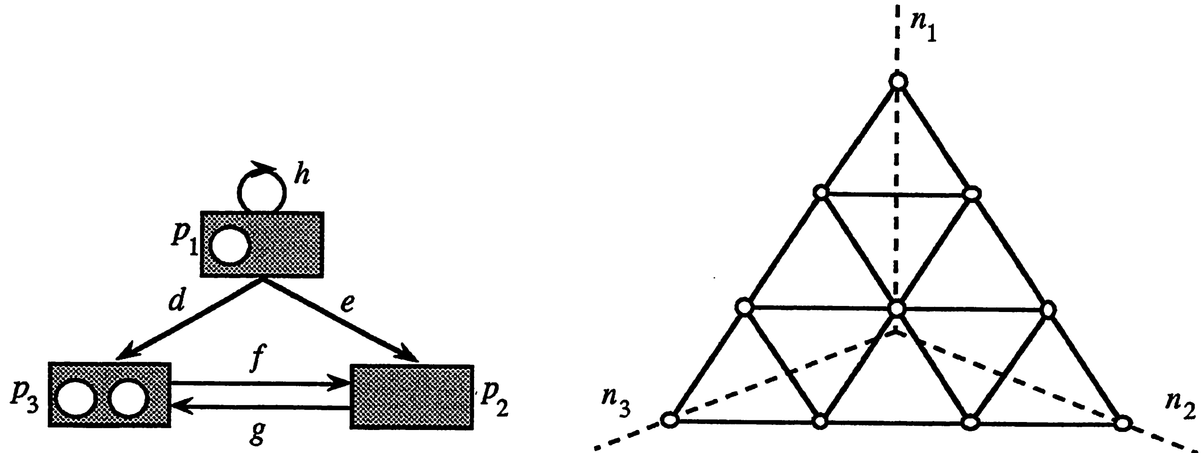


Figure 1: Example of an exchange network and the corresponding simplex $\Delta_{3,3}$

We also associate to each channel $d \in \mathbb{C}$ connecting places p_i to p_j the vector

$$\mathrm{v}_d = (x_1, \ldots, x_k)$$

whose coordinates are all 0 except, when $p_i \neq p_j$, $x_i = -1$ and $x_j = +1$.

Given an initial configuration $\mathcal{C}$ with N tokens, we can construct an associated automaton $\mathbf{A}_{\mathcal{C}}$ on the set of events $\mathbb{C}$, whose states are all accessible configurations from $\mathcal{C}$, and whose transition function is defined by $C \cdot d = C + \mathrm{v}_d$. We say that this automaton "lives" in the simplex $\Delta_{N,k}$. For example, if we take the network of Figure 1 we have the following representation for the associated automaton:

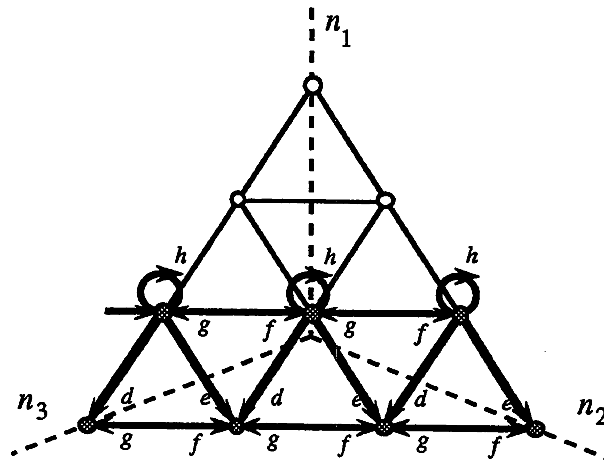


Figure 2: An automaton living in the simplex $\Delta_{3,3}$

Note that if a channel is a loop, the move associated with the channel in a given configuration is

defined iff its source is non-empty (as the move h in Figure 2), and is a loop of the automaton whenever it is defined. When dealing with exchange networks, we will always assume that all states an automaton are final.

Consider a network **E** with places $\mathbb{P} = \{p_1, ..., p_k\}$, channels $\mathbb{C}$, and source and target functions:

$$s, t\colon \mathbb{C} \longrightarrow \mathbb{P}$$

Let $\pi = \{\pi_1, ..., \pi_{k'}\}$ be a partition of $\mathbb{P}$. We can deduce from π a *derived network* $\mathbf{E}_\pi$ with places π, channels $\mathbb{C}$, and source and target functions:

$$s', t'\colon \mathbb{C} \longrightarrow \pi$$

which assign to a channel d the class in π of $s(d)$ and of $t(d)$. The partition π also induces a linear transformation:

$$\mathrm{T}_\pi : \mathbb{N}^k \longrightarrow \mathbb{N}^{k'}$$

defined by

$$\mathrm{T}_\pi(n_1,\dots,n_k) = (\sum_{n_i \in \pi_1} n_i,\dots, \sum_{n_i \in \pi_{k'}} n_i)$$

which maps the simplex $\Delta_{N,k}$ of $\mathbb{N}^k$ onto $\Delta_{N,k'}$ of $\mathbb{N}^{k'}$.

The transformations T_π define morphisms between automata living in simplex: configurations in $\mathbb{N}^k$ are mapped onto configurations in $\mathbb{N}^{k'}$ as if the places in each class π_j were glued together, and vectors associated to moves are mapped onto vectors associated to moves. We have:

Proposition 3.2 Let **E** be a network with places $\mathbb{P}$, initial configuration $\mathfrak{C}$, and associated automaton $\mathbf{A}_\mathfrak{C}$. Let π be a partition of $\mathbb{P}$, and consider the derived network $\mathbf{E}_\pi$ with initial configuration $\mathrm{T}_\pi(\mathfrak{C})$, and associated automaton $\mathbf{A}_{\mathrm{T}_\pi(\mathfrak{C})}$. Then

$$\mathbf{A}_\mathfrak{C} \longrightarrow \mathbf{A}_{\mathrm{T}_\pi(\mathfrak{C})}$$

Proof:

We will show that the function T_π defines a morphism. The two first properties are direct consequences of the definitions of associated automata: $\mathfrak{C}$ and $\mathrm{T}_\pi(\mathfrak{C})$ are the initial states of the two automata, and all states are final.

Suppose now that d connects places p_i to p_j, and that $\pi_{i'}$ and $\pi_{j'}$ are the classes of p_i and p_j in the partition π. Let

$$\mathbf{v}_d = (x_1, ..., x_k) \text{ and } \mathbf{w}_d = (y_1, ..., y_{k'})$$

be the vectors associated to channel d in $\mathbb{N}^k$ and $\mathbb{N}^{k'}$. We have easily that

$$T_\pi(\mathbf{v}_d) = \mathbf{w}_d .$$

If $C\cdot d$ is defined in $\mathbf{A}_{\mathfrak{C}}$, then p_i is not empty in configuration C, thus $\pi_{i'}$ will not be empty in configuration $T_\pi(C)$, thus $T_\pi(C)\cdot d$ is defined. And

$$T_\pi(C)\cdot d = T_\pi(C) + \mathbf{w}_d = T_\pi(C) + T_\pi(\mathbf{v}_d) = T_\pi(C + \mathbf{v}_d) = T_\pi(C\cdot d).$$

■

Let $\mathbf{E}$ be a network with places $\mathbb{P}$, initial configuration $\mathfrak{C}$, and associated automaton $\mathbf{A}_{\mathfrak{C}}$. Consider any sub-automaton $\mathbf{Z} \hookrightarrow \mathbf{A}_{\mathfrak{C}}$. Using Proposition 3.2, we can define the automaton $T_\pi(\mathbf{Z})$ which is the image of $\mathbf{Z}$ by the transformation T_π. And we have $\mathbf{Z} \longrightarrow T_\pi(\mathbf{Z})$. Figure 3 gives an example of such a morphism, with the partition $\{\{p_1, p_2\}, \{p_3\}\}$ of the network in Figure 1.

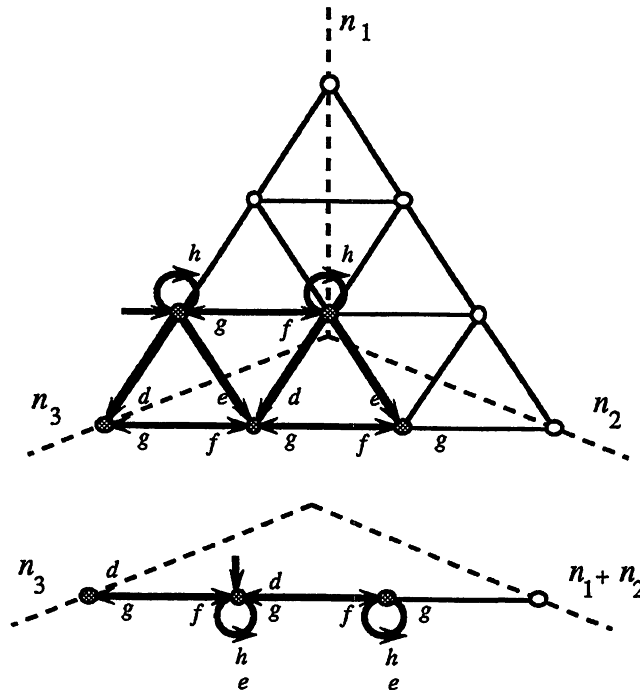


Figure 3: $\mathbf{Z} \longrightarrow T_\pi(\mathbf{Z})$

Linear transformations based on partitions give an elegant way to define observers of specifications. The idea is that a set of unobservable events (channels) defines a natural partition of the places in a network, obtained by identifying places connected by unobservable channels (Figure 4).

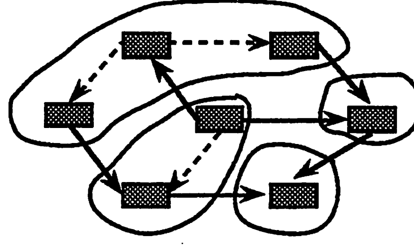


Figure 4: Connected components of the restriction of a network to unobservable channels (dotted arrows are unobservable)

Proposition 3.3 Let **E** be a network with places $\mathbb{P}$, channels $\mathbb{C}$, and $\mathbf{Z} \hookrightarrow \mathbf{A}_{\mathfrak{C}}$ a sub-automaton of the automaton associated to **E** with initial configuration $\mathfrak{C}$. For any subset of channels $\mathbb{U} \subseteq \mathbb{C}$, there exists a partition π of $\mathbb{P}$ such that:

1) $T_\pi(\mathbf{Z})$ is an observation automaton with respect to $\mathbb{U}$,
2) $T_\pi(C)$ counts the number of tokens in each connected component of the restriction of **E** to $\mathbb{U}$.

Proof:

Consider the partition π obtained by identifying places that are in the same connected component of the restriction of the graph **E** to $\mathbb{U}$. Then, if $d \in \mathbb{U}$, $s(d)$ and $t(d)$ are in the same class of the partition, and $T_\pi(v_d)$ is the null vector, establishing that $T_\pi(\mathbf{Z})$ is an observation automaton with respect to $\mathbb{U}$.

The fact that $T_\pi(C)$ counts the number of tokens in each connected component of the restriction of **E** to $\mathbb{U}$ is immediate by construction.

■

4. Networks with bounded capacities

We now turn to a particular class of specifications in exchange networks, *networks with bounded capacities*. These specifications are described with inequalities of the form:

$$C(p_j) \leq \mathrm{Max}_j$$

That is, there is a maximum (> 0) number of tokens allowed in each place. A *legal* configuration is a configuration that satisfies all these inequalities, a *legal* move is a move that links two legal configuration.

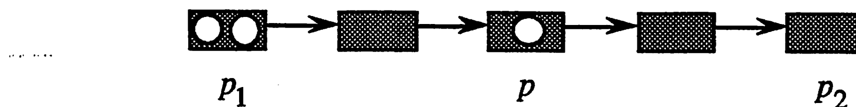
In this section we prove that, if a network satisfies certain connectedness conditions, observability of this kind of specification is *equivalent* to linear observability, thus ensuring efficient computation of observers.

We first establish the following lemma:

Lemma 4.1 If a network with bounded capacities is strongly connected, and C_1, C_2 are two legal configurations, then C_2 is reachable from C_1 by a sequence of legal moves.

Proof:

The proof is based on the following observation. Given two places, p_1 and p_2 such that p_1 is not empty and p_2 is not full, it is possible transfer a token from p_1 to p_2 while keeping invariant the rest of the configuration. Indeed, since the network is strongly connected, there exists a path of channels connecting p_1 to p_2:



p_1 p p_2

Working from the right hand side, we find the first non-empty place p (which exists since p_1 is not empty) at the left of p_2 and transfer one chip from it to p_2. These moves are always possible and legal since the empty places between p and p_2 have capacity at least 1. We then repeat this kind of move until one of the chip in p_1 has finally been moved to p_2.

The general argument is now easy. Assuming that two legal configurations have the same number of tokens, each 'extra' chip in a place of the first one will correspond to a 'hole' in the second one, and vice-versa.

■

Definition 4.2 *Invertible sets of channels*

A set $\mathbb{U} \subseteq \mathbb{C}$ of channels in a network is *invertible* if, in the restriction of the network to $\mathbb{U}$, every connected component is strongly connected.

■

The term invertible comes from the fact that if a token is moved along a channel $d \in \mathbb{U}$, there exists a sequence of channels in $\mathbb{U}$ such that the token can be moved back through them to its original position. The following lemma says that if unobservable channels are invertible for a given site, then any two configuration that are considered the same by an observer are linked by unobservable legal moves:

Lemma 4.3 Let $\mathbb{U}$ be an invertible set of channels in a network **E** with bounded capacities, and π the partition that identifies places connected by channels in $\mathbb{U}$. If C_1, C_2 are two legal configurations such that

$$T_\pi(C_1) = T_\pi(C_2)$$

then there exists a sequence of legal moves in $\mathbb{U}$ connecting C_1 to C_2.

Proof:

By Proposition 3.3, $T_\pi(C)$ counts the number of tokens in each (strongly) connected component of the restriction of the network to the channels in $\mathbb{U}$. Thus if $T_\pi(C_1) = T_\pi(C_2)$, then both configurations have the same number of tokens in each component. Applying Lemma 4.1 to each of these component yield the desired sequence of legal moves in Σ.

■

We are now in position to prove the main theorem of this section. It states that when unobservable events are invertible, observability is equivalent to linear observability in networks with bounded capacities:

Theorem 4.4 Let **Z** be a specification of a network **E** with bounded capacities. Given *n* sites with invertible unobservable events $\mathbb{U}_i$. Let $T_i(\mathbf{Z})$ be the observer at site *i* obtained with the partition induced by $\mathbb{U}_i$. Then

Z is observable ⇔
Z is linearly observable ⇔
Z is linearly observable by $\Pi T_i(\mathbf{Z})$.

Proof:

Clearly, if **Z** is linearly observable by $\Pi T_i(\mathbf{Z})$, then it is observable. In order to show that observability implies linear observability, we will show that observability implies

$$T_i(\mathbf{Z}) \longleftrightarrow \mathbb{O}_i(\mathbf{Z})$$

then, since $\mathbf{Z} \longrightarrow T_i(\mathbf{Z})$, applying Theorem 2.6 with the morphisms $\mathbf{Z} \longrightarrow \mathbb{O}_i(\mathbf{Z})$, we deduce that **Z** is linearly observable by $\Pi\mathbb{O}_i(\mathbf{Z})$, thus by $\Pi T_i(\mathbf{Z})$.

We first show that

$$\mathbb{O}_i(\mathbf{Z}) \longrightarrow T_i(\mathbf{Z})$$

by defining, for any state **S** of $\mathbb{O}_i(\mathbf{Z})$, and any configuration $C \in \mathbf{S}$,

$$f(\mathbf{S}) = T_i(C)$$

For this function to be well defined, we have to show that if $C_1, C_2 \in \mathbf{S}$ then $T_i(C_1) = T_i(C_2)$. This is true for the initial state **I** of $\mathbb{O}_i(\mathbf{Z})$ defined by:

$$\mathbf{I} = \{\ \mathfrak{C}\cdot u \mid \mathfrak{C}\cdot u \text{ is defined and } u \in \mathbb{U}_i{}^*\}$$

since any configuration in this set is of the form $\mathfrak{C}\cdot u$ and $T_i(\mathfrak{C}\cdot u) = T_i(\mathfrak{C})$. Suppose now that the statement is true for a state S. If d is any event such that $S\circ d$ is defined, then $S\circ d$ is either S or

$$S\circ d = \{C\cdot du \mid C \in S,\ C\cdot du \text{ is defined and } u \in \mathbb{U}_i^*\}.$$

If $C_1, C_2 \in S\circ d$, they can be written as

$$C_1 = C_1'\cdot du_1 \text{ and}$$
$$C_2 = C_2'\cdot du_2,$$

where $C_1', C_2' \in S$, and $T_i(C_1') = T_i(C_2')$. We then easily check that:

$$T_i(C_1) = T_i(C_1'\cdot du_1) = T_i(C_1') + T_i(v_d) = T_i(C_2') + T_i(v_d) = T_i(C_2).$$

Thus, the function f is well defined. It induces a morphism since, if $S\circ d$ is defined, then there exists a $C \in S$ such that $C\cdot d$ is defined, thus $T_i(C)\cdot d$ is also defined. And we have:

$$f(S\circ d) = T_i(C\cdot d) = T_i(C)\cdot d = f(S)\cdot d.$$

In order to show that

$$T_i(\mathbf{Z}) \longrightarrow \mathbb{O}_i(\mathbf{Z})$$

we consider the function g defined as the inverse image of the transformation T_i:

$$g(T_i(C)) = T_i^{-1}(T_i(C)).$$

To prove that $T_i^{-1}(T_i(C))$ is always a state of $\mathbb{O}_i(\mathbf{Z})$, we first note that any state S of $\mathbb{O}_i(\mathbf{Z})$ is contained in such a set. Indeed, we know from the first part of the proof that $C_1, C_2 \in S$ implies $T_i(C_1) = T_i(C_2)$ thus

$$S \subseteq T_i^{-1}(T_i(C)) \text{ for any } C \text{ in } S.$$

Now, if $C_1 \in T_i^{-1}(T_i(C))$ and $C \in S$, then $T_i(C_1) = T_i(C)$ and, by Lemma 4.3, there is a sequence of unobservable events that connects C to C_1, implying $C_1 \in S$. So, for any C in S,

$$S = T_i^{-1}(T_i(C)).$$

The function g is thus well-defined. To prove that it induces a morphism we first remark that if $T_i(C)\cdot d$ is defined, then there is a configuration C' *in* $T_i^{-1}(T_i(C))$ such that $C'\cdot d$ is defined, implying that $T_i^{-1}(T_i(C))\circ d$ is defined in $\mathbb{O}_i(\mathbf{Z})$. Furthermore, since $T_i(C) = T_i(C')$, we have

$$g(T_i(C)\cdot d) = g(T_i(C')\cdot d) = g(T_i(C'\cdot d)) = T_i^{-1}(T_i(C'\cdot d))$$

and since $C' \cdot d \in T_i^{-1}(T_i(C)) \circ d$,

$$T_i^{-1}(T_i(C' \cdot d)) = T_i^{-1}(T_i(C)) \circ d = g(T_i(C)) \circ d.$$

■

5. References

Arnold, A., MEC: a system for constructing and analyzing transition systems, In (J. Sifakis, ed.) Automatic verification of finite state systems. Lecture Notes in Computer Science, vol 407, 1989, pp. 117-132.

Arnold, A., Systèmes de transitions finis et sémantique des processus communicants, T.S.I., vol. 9, nº 3, 1990, pp. 193-216.

Bergeron, A., A Unified Approach to Control Problems in Discrete Event Processes, R.A.I.R.O. Informatique Théorique , vol. 27, no 6, 1993, pp. 555-573.

Ramadge, P., Wonham, W., The Control of Discrete Event Systems, Proceedings of the IEEE, vol. 77, nº 1, January 1989, pp. 81-98.

Rudie, K., Wonham, W., Think Globally, Act Locally: Decentralized Supervisory Control, IEEE Transactions on Automatic Control, Vol. 37, nº 11, November 1992, pp. 1692-1708.

Rudie, K., Willems, J., The Computational Complexity of Decentralized Discrete-Event Control Problems, IMA Preprint Series #1105, March 1993.

Tsitsiklis, J., On the Control of Discrete-Event Dynamical Systems, Mat. Control Signals Systems (1989) 2, pp. 95-107.

Yong, L., Wonham, W., Control of Vector Discrete-Event Systems I- The Base Model, IEEE Transactions on Automatic Control, Vol. 38, nº 8, August 1993, pp. 1214-1227.

On Controlling Distributed Communicating Systems

A. Khoumsi* G.v. Bochmann R. Dssouli

Université de Montréal
Faculté des arts et des sciences
Département d'informatique et
de recherche opérationnelle
C.P. 6128, Succursale Centre-Ville
Montréal, (Québec) H3C 3J7

April 1994

Abstract

In this paper, we propose a new approach for controlling a distributed communicating system (DCS). The aim of the control is to restrict the behaviour of the system for obtaining a desirable behaviour. Traditionally, the control consists in forbidding, when desired, the occurrences of some events. In our study, we consider the fact that a distributed system is necessarily composed by several subsystems which, besides interacting with the environment, communicate with each other via a medium of communication. Therefore, the task the local controllers is not only to prevent the occurrences of some events, but also to exchange a private information via the medium of communication. This new approach is applied to a particular structure of distributed sequential communicating systems.

1 Introduction

A discrete event system (DES) is a dynamic system in which events occur instantaneously, causing a discrete change of the state of the system. In this paper, we consider the case where the sequences of events constitute a regular language. Thus, a DES can be modeled by a FSM. Since DESs need in general to be controlled in such a way to avoid several undesirable sequences of events, a control theory was initiated by Ramadge and Wonham ([12, 8]). This theory has subsequently been extended to encompass other aspects, such as decentralized control which interests us in this study ([2, 9]). This extension arises to consider distributed DESs, while the initial theory failed to resolve problems for networks of communicating processes, which can be modeled as distributed DESs

*Supported by FCAR-NSERC-BNR grant

Among the most interesting works about controlling distributed discrete event systems, there are those presented in [2, 9]. Since the results in [9] are the most general, we consider only this reference. Traditionally, the task of the local controllers is to prevent, when desired, the occurrences of some events. In our study, we generalize the task of the local controllers. For that, we use the fact that a distributed system is composed by several local centralized subsystems which interact with the environment and communicate with each other via a medium supposed reliable. Consequently, in our case the task of the local controllers is to :

- Prevent, when desired, the occurrences of some events (traditional task);
- Exchange a private information with each other via the reliable medium (new task).

The remaining of this paper is organized as follows. Section 2 reviews the part of supervisory control framework needed for our study. Section 3 introduces the structure of the distributed communicating systems considered in our study. This structure is based on service and protocol concepts. In Section 4, the new approach of control is detailed and applied to the structure presented in Section 3. A simple example is given in Section 5. Finally, in Section 6, we conclude and propose some possible extensions. Let's notice that the terms "controller" and "supervisor" will be used as synonyms.

2 Control of discrete event systems

2.1 Centralized control

For a given DES noted M_1 and specified by a FSM S_1, the aim of the control is the following. From a desirable behaviour M_0 specified by a FSM S_0, we have to synthesize systematically the controller noted M_2 such that $M_1 \parallel M_2$ —i.e., M_1 working in parallel and in interaction with M_2— behaves as desired. To achieve the desired behaviour, i.e., to influence the evolution of M_1, the controller has to ([5, 8, 12]):

- *track* the evolution of M_1, by observing occurrences of its events;
- *disable* —i.e., prevent occurrences of— some events when desired.

We consider here only the case where all events of M_1 are observable by M_2, i.e., the supervisor detects occurrences of all events of M_1. Before continuing, let's give the following definitions.

Definition 1 A FSM is defined by $S = (Q, K, \delta, q_0)$ where : **(a)** Q is the set of states; **(b)** K is the alphabet, i.e., the set of events; **(c)** δ defines the transitions, i.e., $\delta : Q \times K \longrightarrow Q$, and $\delta(q, \sigma)!$ means that $\delta(q, \sigma)$ is defined; **(d)** q_0 is the initial state.
Let's notice that δ is also defined for a sequence s of events, i.e., $\delta(q, s)$ is the state reached from the state q after the occurrence of the sequence s, and $\delta(q, s)!$ means that $\delta(q, s)$ is defined.

Definition 2 Let $A = (Q_A, V, \delta_A, q_{A0})$ and $B = (Q_B, V, \delta_B, q_{B0})$ be two FSMs defined over a same alphabet V. A is smaller than B (or B is bigger than A), if all sequences of events executable in A from q_{A0}, are executable in B from q_{B0}. This is noted $A \leq B$. In other words, A is smaller than B if and only if the language L_A accepted by A is included in the language L_B accepted by B.

Definition 3 Let A and B be two FSMs over a same alphabet K, respectively accepting the regular languages L_A and L_B. The sum of A and B, noted $A + B$, is the minimal FSM which accepts the language $L_A \cup L_B$.

Since M_2 has to disable some events of M_1, it is natural to partition the set K of events into *controllable* and *uncontrollable* events: $K = K_{co} \cup K_{uc}$ (Figure 1). A controllable event σ ($\sigma \in K_{co}$) is an event whose occurrence can be prevented by M_2. On the contrary, an uncontrollable event γ ($\gamma \in K_{uc}$) is always enabled by M_2. The DES M_1 to be controlled is then modeled by a FSM $S_1 = (Q_1, K, \delta_1, q_{10})$, and the desired behaviour M_0 is specified by a FSM $S_0 = (Q_0, K, \delta_0, q_{00})$, with $K = K_{co} \cup K_{uc}$.

The desired behaviour specified by S_0 is realizable if and only if $S_0 \leq S_1$ (Def. 2) and S_0 does not necessitate to disable uncontrollable events from S_1. In this case, such behaviour is said *controllable*, w.r.t. S_1. Otherwise, it is said *uncontrollable*, w.r.t. S_1. Let then $Cont_{S_1}(S_0)$ be the set of FSMs which specify the controllable behaviours, w.r.t. S_1, which are smaller than or equal to S_0. Since $Cont_{S_1}(S_0)$ is closed under FSM sum (Def. 3, [12]), we can define the supremal element of $Cont_{S_1}(S_0)$, noted $sup(Cont_{S_1}(S_0))$, which is the sum of all elements of $Cont_{S_1}(S_0)$. Algorithms for computing $sup(Cont_{S_1}(S_0))$ can be found in [5, 12]. Let's notice that if S_0 is controllable, w.r.t. S_1, then $S_0 = sup(Cont_{S_1}(S_0))$.

For obtaining the supremal behaviour specified by $sup(Cont_{S_1}(S_0))$, the supervisor M_2 has to observe the evolution of M_1 and to update the set of allowed events. Therefore, M_2 is specified by $C = (S_2, \Psi)$, where :

- $S_2 = sup(Cont_{S_1}(S_0)) = (Q_2, K, \delta_2, q_{2,0})$.
- $\Psi : Q_2 \longrightarrow 2^K$. For each state q of Q_2, $\Psi(q)$ is the set of events of K, which are enabled by M_2 when M_1 has executed a sequence s from its initial state $q_{1,0}$, such that $q = \delta_2(q_{2,0}, s)$. $\Psi(q)$ contains necessarily K_{uc}, and S_2 is smaller than both S_1 and S_0.

In our case where all events are observable—and then the alphabets of S_1 and S_2 are equal—, computing Ψ from S_2 is self-evident. In fact, if $S_2 = (Q_2, K, \delta_2, q_{2,0})$ then Ψ is formally defined by ([5]) :

$$\forall q \in Q_2 : \Psi(q) = K_{uc} \cup \{\sigma | (\sigma \in K_{co}) \wedge (\delta_2(q, \sigma)!)\}.$$

Example 1 The alphabet of S_1 and S_0 of Figures 2.(a) and 2.(b) is $K = K_{co} \cup K_{uc}$, where $K_{co} = \{a, b, c, d, e\}$ and $K_{uc} = \{\alpha, \beta\}$. S_0 is uncontrollable since it necessitates to disable the uncontrollable events α and β, respectively at states 3 and 4 of S_1. By using the algorithm proposed in [5], we compute the FSM $S_2 = sup(Cont_{S_1}(S_0))$ represented on Figure 2.(c). The behaviour of the controlled M_1 is modeled by S_2 if the supervisor disables the event b at state 1, and the event d at state 2. Therefore : $\Psi(1) = K \setminus \{b\} = \{a, c, d, e\} \cup \{\alpha, \beta\}$, $\Psi(2) = K \setminus \{d\} = \{a, b, c, e\} \cup \{\alpha, \beta\}$, and the supervisor is specified by $C_2 = (S_2, \Psi)$.

2.2 Decentralized control

A distributed communicating system (DCS) is a DES whose events may occur in different sites. The set K of events is then partitionned into $K_1, K_2, \ldots K_n$, where n is the number of sites, and K_i is the set of events occurring in site i. We consider here only the case where : $K = \cup_{i=1}^{n} K_i$, and $\forall i, j \leq n : i \neq j \Rightarrow K_i \cap K_j = \emptyset$.

Intuitively, this means that the different sites are disjoint (Fig. 3). Each element of the alphabet K is defined by e_i, where e is the name of an action and i identifies the site where e occurs. Let's notice that the events of K correspond to both: **(a)** interactions between the DCS and the environment; **(b)** communication between the sites. This fact is detailed in Section 3.

For controlling a DCS M_1, n local supervisors $M_{2,1}, \ldots, M_{2,n}$ may be necessary. Each $M_{2,i}$ can observe all and only the events of K_i and can disable only controllable events of K_i. Therefore, each K_i is partitionned into $K_{i,co}$ and $K_{i,uc}$, i.e., $K_i = K_{i,co} \cup K_{i,uc}$, which respectively represent controllable and uncontrollable events on site i. We also define $K_{co} = \cup_{i=1}^{n} K_{i,co}$, $K_{uc} = \cup_{i=1}^{n} K_{i,uc}$, and then $K = K_{co} \cup K_{uc}$. For a DCS M_1 modeled by a FSM S_1, the aim of a decentralized control is then the following one. From a global desirable behaviour M_0 specified by a FSM S_0, we have to synthesize systematically the local controllers $M_{2,i}$, for $i = 1, \ldots, n$, such that M_1 behaves as desired when it is in interaction with the n local supervisors. Before continuing, let's define different kinds of projections.

Definition 4 Projections
Def. 4.1: (*Projection of an event*). Let σ be an event of the alphabet K. The projection of σ on an alphabet K_i is noted $P_i(\sigma)$ and is defined by :

$$P_i(\sigma) = \begin{cases} \sigma & \text{if } \sigma \in K_i \\ \epsilon & \text{otherwise} \end{cases}$$

where ϵ is an empty sequence (without event).

Def. 4.2: (*Projection of a sequence*). Let s be a finite sequence of events, and let σ be an event. The projection of a sequence on an alphabet K_i is defined recursively by:

$$\begin{cases} P_i(\epsilon) = \epsilon \\ P_i(s\sigma) = P_i(s)P_i(\sigma) \end{cases}$$

Example: if $K = \{a, b, c\}$, $K_1 = \{a, b\}$ and $s = acbabcbcca$, then $P_1(s) = ababba$.

Def. 4.3: (*Projection of a regular language*). If L is a regular language, the projection of L on K_i is defined by: $P_i(L) = \{P_i(s) \mid s \in L\}$.

Def. 4.4: (*Projection of a FSM*). Each FSM A accepts a regular language noted L_A. The projection of A on an alphabet K_i is the minimal FSM noted $P_i(A)$ which accepts the language $P_i(L_A)$. In other words : $L_{P_i(A)} = P_i(L_A)$.

For obtaining a desired and realizable behaviour specified by a FSM S_0, each supervisor $M_{2,i}$ has to :**(a)** observe the local evolution of M_1, i.e., the occurrences of events of M_1 in

site i; **(b)** update the set of local allowed events, i.e., the set of events which may occur in site i. Therefore, each $M_{2,i}$ is specified by $C_i = (S_{2,i}, \Psi_i)$, where :

- $S_{2,i} = P_i(S_0) = (Q_{2,i}, K_i, \delta_{2,i}, q_{2,i,0})$;
- $\Psi_i : Q_{2,i} \longrightarrow 2^{K_i}$ is defined by $\Psi_i(q) = K_{i,uc} \cup \{\sigma \mid (\sigma \in K_{i,co}) \wedge (\delta_{2,i}(q, \sigma)!)\}$.

Intuitively, when M_1 executes the sequence s and is then in state $q_1 = \delta_1(q_{1,0}, s)$, each supervisor $M_{2,i}$, for $i = 1, \ldots, n$, is in state $q_2 = \delta_{2,i}(q_{2,i,0}, P_i(s))$ and enables events in $\Psi_i(q_2)$.

In [9], it is proven that a desired behaviour S_0 is realizable if and only if S_0 is controllable and n-observable, w.r.t. S_1. The controllability is defined in Section 2.1, with $K_{co} = \cup_{i=1}^{n} K_{i,co}$ and $K_{uc} = \cup_{i=1}^{n} K_{i,uc}$. A formal definition of n-observability is given in [9], and here we give only an intuitive idea. A behaviour S_0 is n-observable if it necessitates that the local supervisors have to make decisions which depends only on what they observe. In other words, every local supervisor $M_{2,i}$ takes a same decision after the executions, from the initial state of S_1, of two sequences s and t such that $P_i(s) = P_i(t)$. If n=2, the n-observability is called coobservability ([9]). If there is only one local supervisor, the n-observability is equivalent to the observability ([5, 8]).

Example 2 The alphabet of S_1 and S_0 of Figures 4.(a) and 4.(b) is $K = K_1 \cup K_2$ with $K_1 = K_{1,co} = \{a_1, b_1\}$, $K_2 = K_{2,co} \cup K_{2,uc}$, $K_{2,co} = \{c_2\}$ and $K_{2,uc} = \{\delta_2\}$. S_0 is not controllable because it necessitates to disable the uncontrollable event δ_2 from state 3. S_0 is not coobservable because $M_{2,2}$ (supervisor in site 2) must enable δ_2 after the occurrence of event a_1 (State 2), and disable the same event δ_2 after the occurrence of sequence a_1b_1 (State 3). This is not possible because $M_{2,2}$ cannot know if M_1 is at state 1,2 or 3, since $P_2(a_1) = P_2(a_1b_1) = \epsilon$. The supremal controllable behaviour $S_2 = sup(Cont_{S_1}(S_0))$ is represented on Figure 4.(c). In this example, S_2 is realizable because it is controllable and *coobservable*. The local supervisor $M_{2,1}$ has to disable b_1, while $M_{2,2}$ has to enable c_2 and δ_2 (Fig. 4.(f)). $M_{2,1}$ and $M_{2,2}$ are respectively specified by $C_1 = (S_{2,1}, \Psi_1)$ and $C_2 = (S_{2,2}, \Psi_2)$. $S_{2,1}$ and $S_{2,2}$ are represented on Figures 4.(d) and 4.(e), $\Psi_1(1) = \{a_1\}$ and $\Psi_2(1) = \{c_2, \delta_2\}$. In this example, $M_{2,2}$ is not really necessary since it disables no event. Therefore, in general if we obtain a local supervisor $M_{2,i}$ which enables all local events of K_i, then $M_{2,i}$ is not necessary.

In the next section, we introduce the structure of the distributed communicating systems considered in our study. Such structure is based on service and protocol concepts. Afterwards in Section 4, we propose a procedure for controlling a DCS having the structure considered. With this procedure, the local supervisors $M_{2,i}$, not only prevent occurrences of some events, but they also exchange some private information with each other. In this case, the controllable behaviours, which are not realizable by only preventing occurrences of events, become realizable.

3 Distributed communicating structure

3.1 Introduction

A distributed communicating system is a system shared on different sites which can communicate :

- with the user (environment) via servive access points (SAP)
- with each other via a medium.

The medium is supposed reliable since it is not just a physical link, but it also contains all software and hardware tools necessary to hide the unreliability of the physical link. With the 7-layer OSI architecture, the medium provides at least a service of the transport layer ([11]). Therefore, a message sent from a site i to a site j, reaches its destination without being corrupted. Let's notice that the term "user" is used in a general case, i.e., the user represents the environment which interacts with the DCS.

3.2 Service and protocol concepts

Each site i contains a module, called protocol entity and noted PE_i, which : **(a)** interacts with the user of the site i; **(b)** communicates with the medium, to exchange messages with other protocol entities. In the user's viewpoint, the distributed system is globally a black box, where interactions with the medium are invisible (Fig. 5.(a)). Therefore, the specification of the service provided to the user (called *service specification*) defines the ordering of the interactions visible by the user. These interactions are called service primitives. Informally, such specification defines the service provided to (or desired by) the user of the DCS.

In the designer's viewpoint , it is necessary to generate the local specifications of the n protocol entities (Fig. 5.(b)), $PE_1, \ldots, PE_n$ (called ***protocol specifications***) from the specification of a desired service. Informally, each local specification of PE_i specifies "what is implemented in site i". An approach which directly generates protocol specifications is called synthesis ([1, 3, 4, 6, 7, 10]). In our present study, we use Finite automata as a formalism of specification, and we consider only sequential systems.

Definition 5 A sequential service is described by a FSM $SS = (Q_s, K_s, \delta_s, q_{s,0})$ which specifies the global ordering of service primitives (SP) observed by the user at the different sites. Events of the alphabet K_s of SS are noted e_i, where e is the name of a service primitive (SP), and i identifies the site where the SP is executed. The occurrence of an event e_i means that the service primitive e is executed in the site i by the protocol entity PE_i.

Example 3 Let the distributed system constituted by two sites and schematized on Figure 6.(a). A formal specification of the service is represented on Figure 6.(b). Four service primitives a, b, c and d are defined, and the alphabet is $K_s = \{a_1, b_1, c_1, d_1, c_2, d_2\}$,

Definition 6 A protocol entity PE_i is described by a FSM $PS_i = (Q_i, K_i, \delta_i, q_{i,0})$ which specifies the ordering of the local interactions with the user and with the medium on site i. The elements of the alphabet K_i, i.e., the events which occur in site i, are of three types.

1. Execution of a service primitive e on site i (interaction with the local user). This event is noted e_i.

2. Sending a message with a parameter p from PE_i to a protocol entity PE_j. This event is noted $s_i^j(p)$.

3. Reception by PE_i of a message with a parameter p coming from a protocol entity PE_j. This event is noted $r_i^j(p)$.

The use of the parameter p contained in a message is explained in Section 3.3. An example of protocol specifications is given on Figure 8 (See Example 4 in Section 3.3).

3.3 Synthesizing the protocol from the service

Before introducing the principle of the procedure for synthesizing the protocol, let's define the global protocol specification.

Definition 7 The global protocol specification is a FSM $GPS = (Q, K, \delta, q_0)$ which specifies the ordering of all events (interactions with the user and with the medium) which occur in the distributed system. Therefore : $K_s \subset K = \cup_{i=1}^{n} K_i$. The service and protocol specifications SS and PS_i, for $i = 1, \ldots, n$, can be obtained from GPS by projections (Def. 4 on Section 2). Let P_s and P_i, for $i = 1, \ldots, n$, be respectively the projections on the alphabets K_s and K_i, for $i = 1, \ldots, n$. Then $SS = P_s(GPS)$, and $PS_i = P_i(GPS)$ (Example 4). Informally, GPS specifies the global structure of the distributed communicating system. An example of GPS is given on Figure 7 (See Example 4).

The aim of synthesis is the generation of the protocol specifications PS_i, for $i = 1, \ldots, n$, from the desired service specification SS. Intuitively, we have to generate what must be implemented in each site from what the user desires. The basic principle used for synthesizing the different PS_i is the following ([1, 3, 4, 6, 7, 10]).

- If, in SS, two consecutive service primitives A and B are executed by two different protocol entities PE_i and PE_j, i.e., if in SS two transitions A_i and B_j, with $i \neq j$, are consecutive, then :

  1. PE_i executes A and sends a message to PE_j. This message is parameterized by the identifier p of the state of SS reached after the execution of A.
  2. When PE_j receives the message parameterized by p, it executes B.

- If after a transition A_i, there is a choice between m transitions Bk_{jk} executed by the protocol entities PE_{jk}, for $k = 1, \ldots, m$, then :

  1. PE_i executes the primitive A and selects one of the protocol entities PE_{jk} which executes one of the primitives Bk.

2. PE_i sends a message to the selected PE_{jk}. This message is parameterized by the identifier p of the state of SS reached after the execution of A.
3. When PE_{jk} receives the message parameterized by p, it may execute Bk.

The parameter p contained in a message is necessary to avoid any ambiguity when a protocol entity receives a message. From this basic principle, several systematic methods of synthesis are developped in the literature, and we propose the one used in [7]. The latter generates, in a first step, the global specification GPS from the specification SS of the desired service. In a second step, PS_i ,for $i = 1, \ldots, n$, are obtained by projecting GPS on the alphabets K_i.

Example 4 From the service specification of Figure 6.(b), the obtained GPS is represented on Figure 7. From GPS, we deduce the two alphabets K_1, K_2 and K as follows :
$K_1 = \{a_1, b_1, c_1, d_1, s_1^2(4), r_1^2(2)\}$; $K_2 = \{c_2, d_2, s_2^1(2), r_2^1(4)\}$; $K = K_1 \cup K_2$. The projections of GPS on alphabets K_i, for $i = 1, 2$, give the two protocol specifications PS_i of Figure 8.

4 Controlling a sequential DCS

4.1 Introduction of the problem

Let a sequential DCS (SDCS) M_1 modeled by a FSM GPS_1, over the alphabet K which contains all interactions with the user and the medium. In the user's viewpoint, M_1 can be modeled by a FSM SS_1 which specifies the service provided to the user. The alphabet K_s of SS_1 contains only interactions with the user, and SS_1 is then such that $SS_1 = P_s(GPS_1)$, where P_s is the projection on the alphabet K_s. As an example, let the SDCS M_1 modeled by GPS_1 and SS_1 respectively represented on Figures 7 and 6.(b).

The problem is then the following. From a specification SS_0 which models a desired service,over the alphabet K_s, the aim is to control the SDCS M_1 in such a way that it provides the biggest realizable service which is smaller than SS_0. The entries of the problem are then : GPS_1, SS_1 and SS_0.

4.2 Approach for the problem

A desired behaviour of a distributed system is realizable, by only preventing occurrences of some events, if and only if it is controllable and n-observable ([9],Section 2.2). Intuitively, a behaviour is not n-observable if the decisions of at least one local controller $M_{2,i}$ do not depend only on what $M_{2,i}$ observes locally, but also on previous decisions of local controllers in other sites. In other words, at least one local supervisor needs additional information for making its decisions. As an example, the behaviour specified by the FSM of Figure 4.(b) is neither controllable nor n-observable (see also Example 3 in Section 2.2).

Since there is a possibility for exchanging information between the sites, via a medium of communication, we propose that the different local controllers, besides forbidding some events, exchange information in such a way that every local supervisor receives all the

necessary information for making its decisions. In this case, a controllable behaviour, which is initially *not* n-observable, can be made realizable by adding to it an exchange of information between the local supervisors.

Let's propose a way for exchanging information between the different local supervisors $M_{2,i}$, for $i = 1,\ldots,n$, in order to control a SDCS M_1. Each $M_{2,i}$ interacts with the corresponding protocol entity PE_i of site i by forbidding some local events, and exchanges some private information with other local supervisors. The private information is transmitted by filtering the messages exchanged between the protocol entities. The message filtering is realized as follows (Figure 9).

Let a protocol entity PE_i which has to send to PE_j a message parameterized by p (event $s_i^j(p)$). The message is intercepted by $M_{2,i}$ which adds a private information m in it, before sending the message to PE_j. Such operation is called *Filtering of a transmitted message*, and is noted $s_i^j(p, +m)$.

When the message reaches the site j, it is intercepted by $M_{2,j}$ which removes the private information m, before giving the message to PE_j. Such operation is called *Filtering of a received message*, and is noted $r_j^i(p, -m)$.

4.3 Synthesizing local supervisors

Before proposing a procedure which generates systematically the formal specifications of the local supervisors $M_{2,i}$, let's define the operator $\otimes$.

Definition 8 Let A and B be two FSMs respectively over the alphabets V_A and V_B, and accepting respectively the languages L_A and L_B. Let $A \times B$ be the synchronized product of A and B. If for instance $V_A = V_B$, then $A \times B$ accepts the language $L_A \cap L_B$.
$A \otimes B$ is the supremal (biggest) FSM which is smaller than $A \times B$ and is free of deadlock.

The proposed procedure for synthesizing the local supervisors is noted *Synt_Loc_Sup* and is composed of the five following steps. The entries of *Synt_Loc_Sup* are the FSMs GPS_1, SS_1 and SS_0 (Sect. 4.1). GPS_1 is defined over the alphabet $K = K_s \cup K_m$, where K_s is the alphabet of SS_1 and SS_0, and K_m contains events corresponding to the communication between the sites.

Step 1. If SS_0 is not smaller than SS_1 or contains deadlocks, then SS_0 is replaced by $SS_0 \otimes SS_1$ (Def. 8).

Step 2. The FSM $GPS_{1,0} = GPS_1 \otimes SS_0$ is computed. $GPS_{1,0}$ models the supremal behaviour of M_1 which provides the desired service (because $SS_0 = P_s(GPS_{1,0})$) and is free of deadlocks.

Step 3. $GPS_{1,0}$ is possibly not controllable, and then not realizable. Therefore, the supremal controllable GPS_c, w.r.t. GPS_1, which is smaller than GPS_0 is computed. This is noted $GPS_c = sup(Cont_{GPS_1}(GPS_{1,0}))$ (Section 2). Let's notice that for computing GPS_c, all the events $r_i^j(p)$ are considered uncontrollable.

Step 4. GPS_c is possibly not n-observable (Sections 2.2 and 4.2), and then not realizable. In this case, it may be impossible to provide the service $SS_c = P_s(GPS_c)$. Therefore, GPS_c is transformed into GPS with the following rules. If $E_{i,j}(p)$ is the sequence composed of event $s_i^j(p)$ followed by $r_j^i(p)$ (Figure 10.(a)), then for any i, j and p, such that $E_{i,j}(p)$ is defined in GPS_c :

Case 1. If $E_{i,j}(p)$ is single in GPS_c then it remains unchanged.

Case 2. If all sequences $E_{i,j}(p)$ lead to a same state of GPS_c (Figure 10.(b)), then they remain unchanged.

Case 3. Otherwise, each $E_{i,j}(p)$ leading to a state identified by m (Figure 10.(c)), is replaced by the sequence of $s_i^j(p, +m)$ followed by $r_j^i(p, -m)$ (Figure 10.(d)), where $s_i^j(p, +m)$ and $r_j^i(p, -m)$ are defined in Section 4.2.

This transformation is noted *Filt*, i.e., $GPS = Filt(GPS_c)$, and is such that $P_s(Filt(GPS_c)) = P_s(GPS_c)$. Informally, the transformation *Filt* does not change the provided service. Contrary to GPS_c, the behaviour specified by $Filt(GPS_c)$ is always realizable. Intuitively, the private information exchanged between the local controllers $M_{2,i}$ —$s_i^j(p, +m)$ followed by $r_j^i(p, -m)$— eliminates the reazon why GPS_c is not n-observable. In fact, every $M_{2,i}$ will have all necessary information for deciding which local controllable events are allowed.

Step 5. The specifications of the local supervisors are computed by projecting GPS in the alphabets of the different sites. In other words, for each site i, $M_{2,i}$ is specified by $(S_{2,i}, \Psi_i)$ where :

- $S_{2,i} = P_i(GPS) = (Q_{2,i}, K_{2,i}, \delta_{2,i}, q_{2,i,0})$, P_i being the projection on the set K_i of events occurring in site i.
- $\Psi_i : Q_{2,i} \longrightarrow 2^{K_i}$, where $\Psi_i(q)$ is the set of local events allowed when $M_{2,i}$ is in state q. Formally : $\Psi_i(q) = K_{i,uc} \cup \{\sigma \mid (\sigma \in K_{i,co}) \wedge (\delta_{2,i}(q, \sigma)!)\}$.
  In the definition of $\Psi_i(q)$, $\delta_{2,i}(q, s_i^j(p))!$ (resp. $\delta_{2,i}(q, r_i^j(p))!$) means that the event $s_i^j(p)$ or an event $s_i^j(p, +m)$ (resp. $r_i^j(p)$ or $r_i^j(p, -m)$) is executable from the state q of $S_{2,i}$.

5 Example

Let the SDCS to be controlled modeled by GPS_1 of Figure 7, and then providing the service modeled by SS_1 of Figure 6.(b). The desired service is specified by SS_0 represented on Figure 11.
Procedure *Synt_Loc_Sup*

Step 1. SS_0 is smaller than SS_1 and is free of deadlocks, then $SS_1 \otimes SS_0 = SS_0$

Step 2. The obtained $GPS_{1,0}$ is represented on Figure 12 and is not controllable, because it necessitates to forbid the uncontrollable event b_1 at state 3.

Step 3. The supremal controllable GPS_c, w.r.t. GPS_1, which is smaller than $GPS_{1,0}$, i.e., $GPS_c = sup(Cont_{GPS_1}(GPS_{1,0}))$, is represented on Figure 13. In this example, the controllable events are a_1, c_1, d_1 and all events $s_i^j(p)$. Thus, the uncontrollable events are b_1, c_2, d_2 and all events $r_i^j(p)$.
Intuitively, since the uncontrollable event b_1 is forbidden from state 3 of GPS, then the state 3 must be avoided by forbidding the controllable event d_1 from state 2.

Step 4. GPS_c is not coobservable, w.r.t. GPS_1, and then not realizable (Section 2.2). Intuitively, when the message with parameter 2 is received in site 1 (by $r_1^2(2)$), the local controller $M_{2,1}$ cannot know if it corresponds to the event which leads to state 2 or to state 12 (Figure 13). Therefore, $M_{2,1}$ cannot decide to forbid the controllable event c_1. If we apply the transformation $Filt$ to GPS_c, we obtain $GPS = Filt(GPS_c)$ of Figure 14. Intuitively, $M_{2,1}$ can decide to forbid the event c_1 because the message coming from site 2 contains an information (2 or 12) which did not exist in GPS_c.

Step 5. Each $M_{2,i}$, for $i = 1, 2$, is specified by $(S_{2,i}, \Psi_i)$. The FSMs $S_{2,1}$ and $S_{2,2}$ are represented on Figure 15. Let $K_{i,co}$ and $K_{i,uc}$ be respectively the sets of controllable and uncontrollable events in site i. Therefore, $K_{1,co} = \{a_1, c_1, d_1, s_1^2(4)\}$, $K_{1,uc} = \{b_1, r_1^2(2)\}$, $K_{2,co} = \{s_2^1(2)\}$ and $K_{2,uc} = \{c_2, d_2, r_2^1(4)\}$. The functions Ψ_1 and Ψ_2 are as follows :

$\Psi_1(1) = K_{1,uc} \cup \{a_1\}$, $\Psi_1(2) = K_{1,uc}$, $\Psi_1(3) = K_{1,uc} \cup \{s_1^2(4)\}$, $\Psi_1(4) = K_{1,uc}$, $\Psi_1(5) = K_{1,uc} \cup \{c_1\}$.

$\Psi_2(1) = K_{2,uc}$, $\Psi_2(2) = K_{2,uc}$, $\Psi_2(3) = K_{2,uc}$, $\Psi_2(4) = K_{2,uc} \cup \{s_2^1(2)\}$, $\Psi_2(5) = K_{2,uc} \cup \{s_2^1(2)\}$.

Let's notice that, in this example, the task of $M_{2,2}$ is only to inform $M_{2,1}$ if d_2 has occurred before c_2. If yes, $M_{2,2}$ adds the parameter 12 in the sent message $(s_2^1(2, +12))$. Otherwise, $M_{2,2}$ adds the parameter 2 in the sent message $(s_2^1(2, +2))$.

6 Conclusion

In this study, a new approach is proposed for controlling a distributed communicating system (DCS). Besides preventing occurrences of some controllable events, the task of the local controllers is also to exchange a private information with each other. With this approach, the controllable behaviours of a DCS which are unrealizable when the local controllers do not communicate with other, become realizable. A procedure is proposed for generating automatically the specifications of the local controllers when the DCS to be controlled is sequential. The procedure is applied to a simple example.

For future work, we intend to extend our study by considering the control of distributed systems with *timing requirements*. We also intend to study the control of *concurrent* distributed systems.

References

[1] G.v. Bochmann and R. Gotzhein. Deriving protocols specifications from service specifications. In *Proceedings of the ACM SIGCOMM Symposium*, USA, 1986.

[2] R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, March 1988.

[3] C. Kant, T. Higashino, and G.v. Bochmann. Deriving protocol specifications from service specifications written in lotos+. Technical Report 805, Université de Montréal. Département d'informatique et de recherche opérationnelle, C.P. 6128, Succursale Centre-Ville, January 1992.

[4] F. Khendek, G.v. Bochmann, and C. Kant. New results on deriving protocol specifications from services specifications. In *Proceedings of the ACM SIGCOMM Symposium*, pages 136–145, 1989.

[5] A. Khoumsi, G.v. Bochmann, and R. Dssouli. Contrôle et extension des systèmes à événements discrets totalement et partiellement observables. In *Proceedings of The Third Maghrebian Conference on Software Engineering and Artificial Intelligence*, Rabat, Morocco, April 1994.

[6] A. Khoumsi, G.v. Bochmann, and R. Dssouli. Dérivation de spécifications de protocole à partir de spécifications de service avec des contraintes temps-réel. *Réseaux et Informatique Répartie*, 1994.

[7] A. Khoumsi, G. v. Bochmann, R. Dssouli, and A. Ghedamsi. A systematic and optimized method for designing protocols for real-time applications. Technical Report 900, Université de Montréal. Département d'informatique et de recherche opérationnelle, C.P. 6128, Succursale Centre-Ville, 1994.

[8] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. In *Proceedings of the IEEE*, volume 77, pages 81–98, January 1989.

[9] K. Rudie and W.M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, November 1992.

[10] K. Saleh and R. Probert. A service based method for the synthesis of communicating protocols. *International Journal of Mini and Microcomputer*, 12(3), 1992.

[11] A. Tanenbaum. *Réseaux: Architectures, protocoles, applications.* InterEditions, Paris, 1990.

[12] W.M. Wonham and P.J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM J.Control and Optimization*, 25(3):637–659, May 1987.

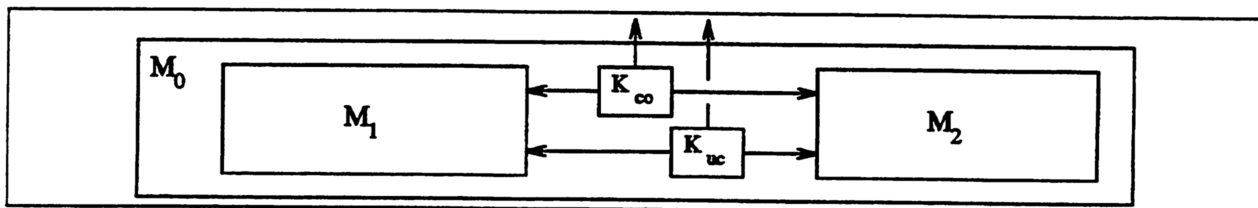


Figure 1: Symbolic representation of controllable and uncontrollable events.

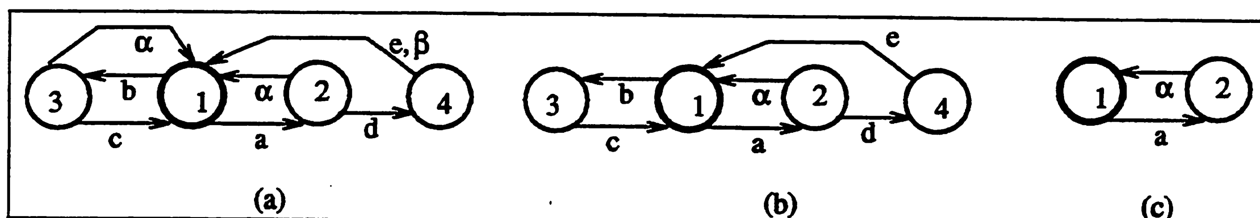


Figure 2: Example of a centralized control. (a) S_1. (b) S_0. (c) $S_2 = sup(Cont_{S_1}(S_0))$.

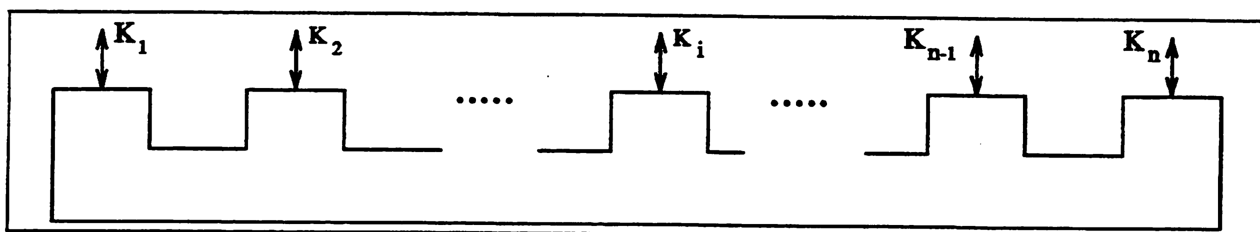


Figure 3: Symbolic representation of a distributed system.

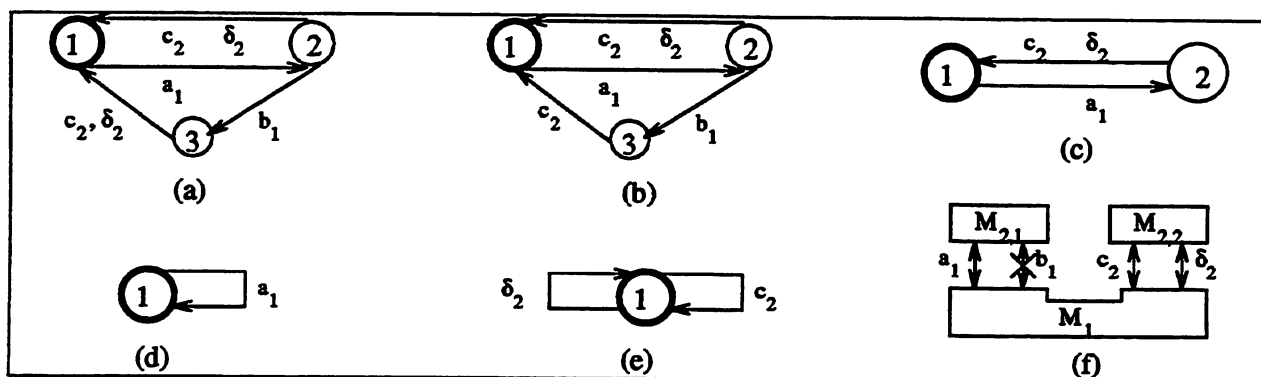


Figure 4: Example of a decentralized control problem. (a) S_1. (b) S_0. (c) $S_2 = sup(Cont_{S_1}(S_0))$. (d) $S_{2,1}$. (e) $S_{2,2}$. (f) Symbolic representation of local supervisors.

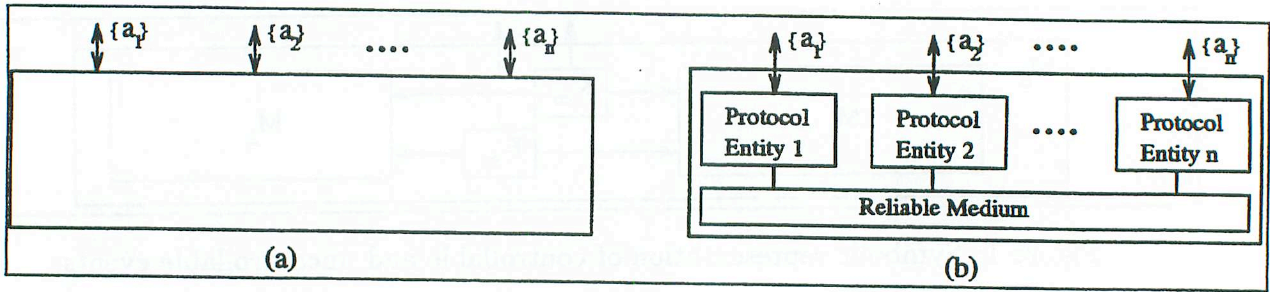


Figure 5: Service and protocol concepts. (a) Service. (b) Protocol.

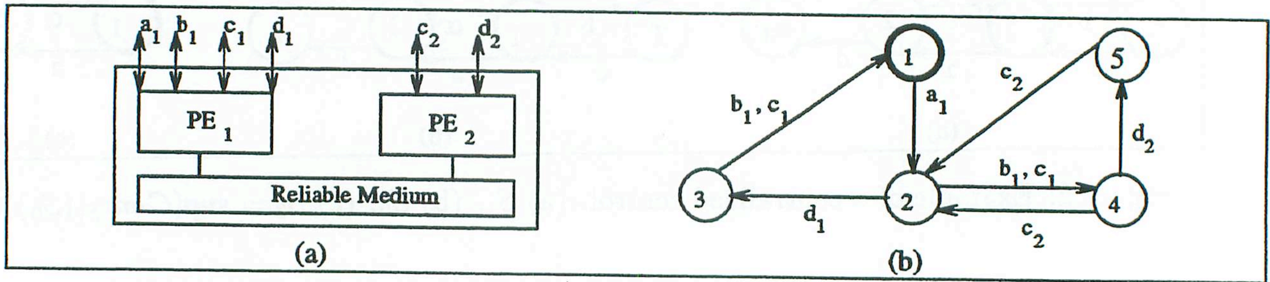


Figure 6: Example of a service. (a) DCS with two sites. (b) Service Specification.

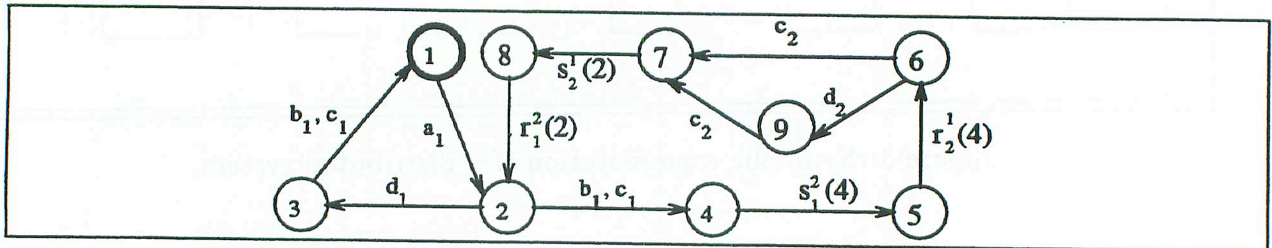


Figure 7: Example of a global protocol specification.

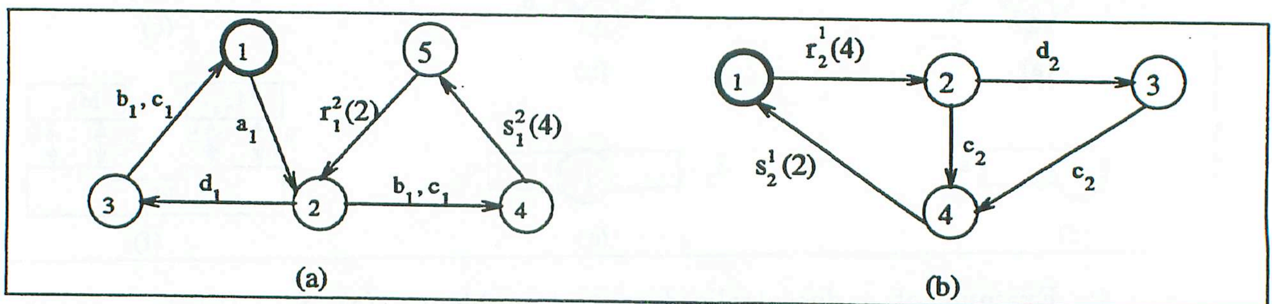


Figure 8: Synthesized protocol specifications. (a) PS_1. (b) PS_2.

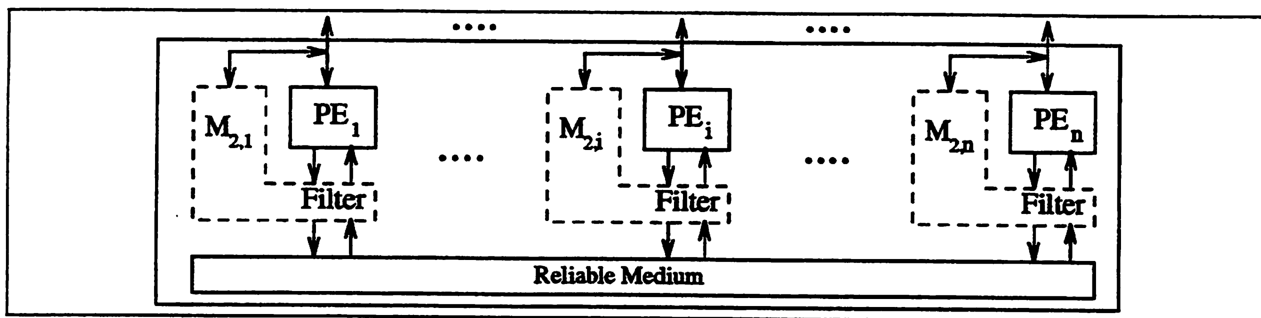


Figure 9: Local controllers.

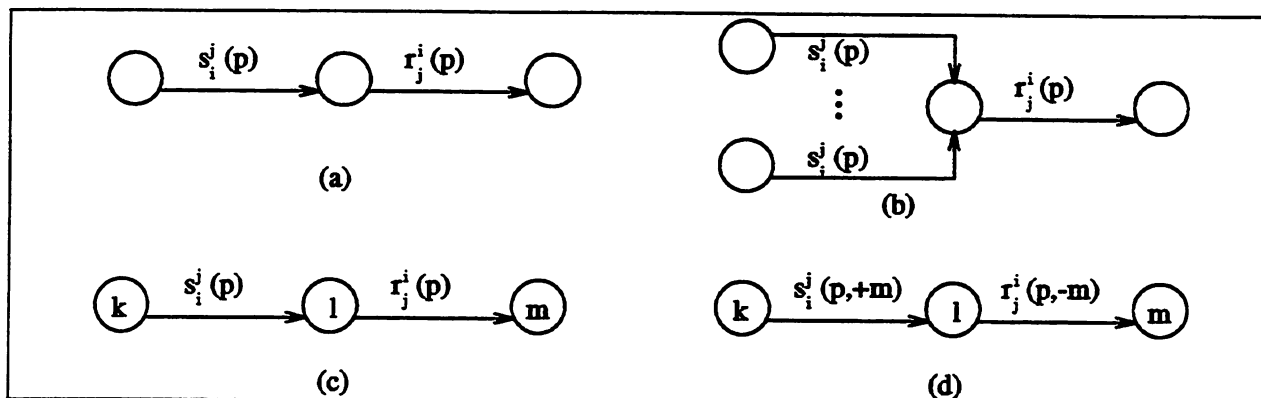


Figure 10: Rules of filtring.

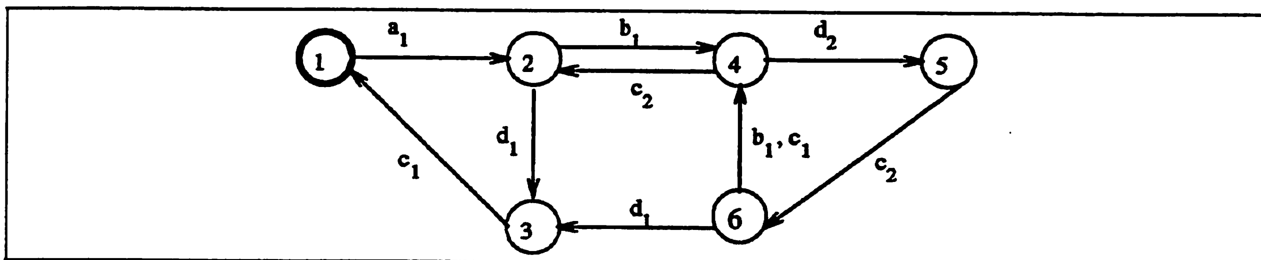


Figure 11: Example of a desired service specification SS_0.

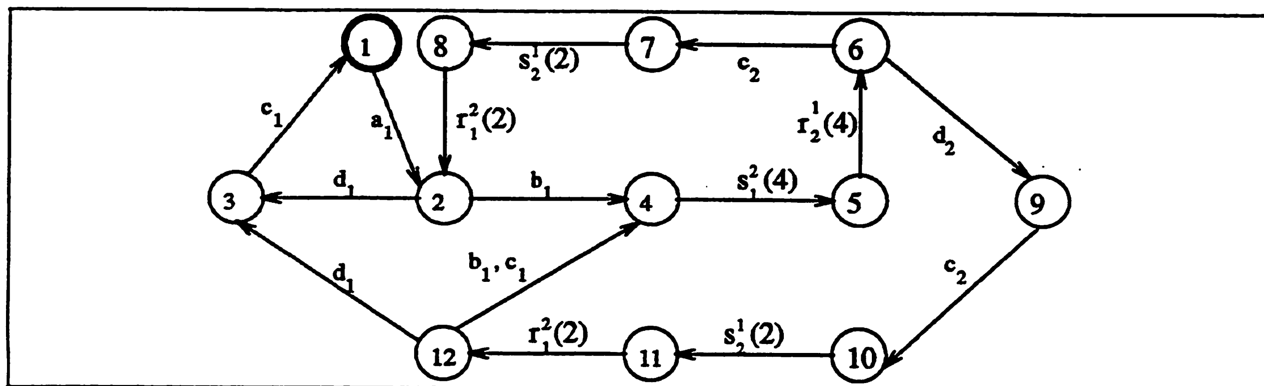


Figure 12: $GPS_{1,0}$ (obtained at step 2 of *Synt_Loc_Sup*).

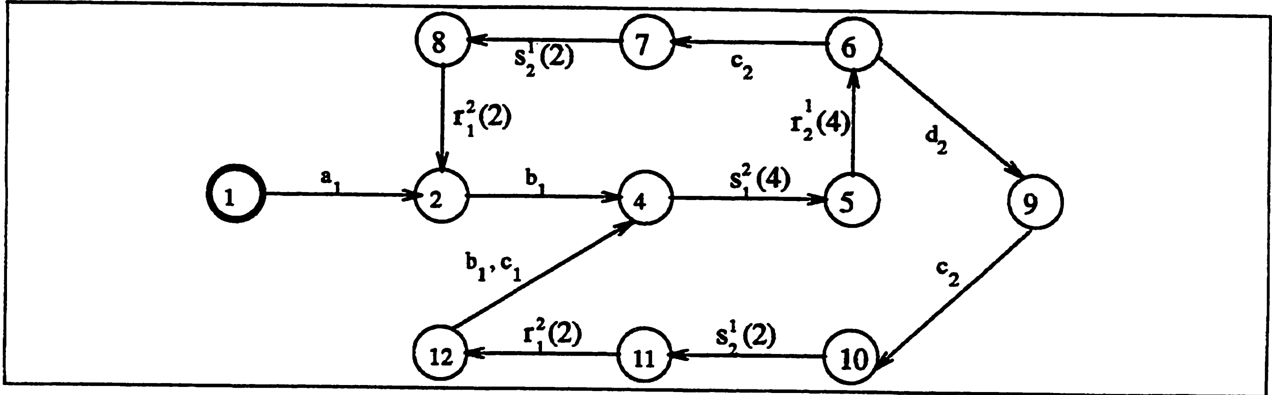


Figure 13: The supremal controllable GPS_c (obtained at step 3 of *Synt_Loc_Sup*).

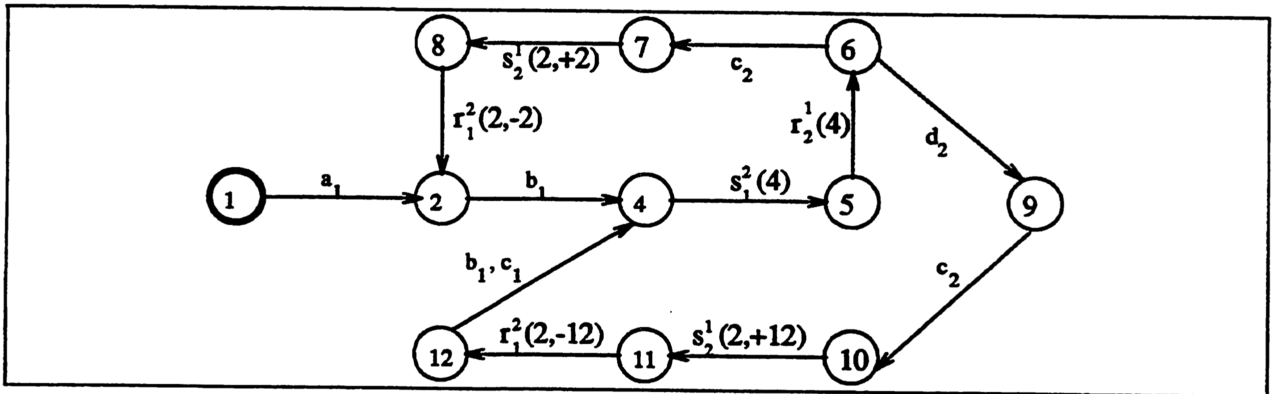


Figure 14: $Filt(GPS_c)$ (obtained at step 4 of *Synt_Loc_Sup*).

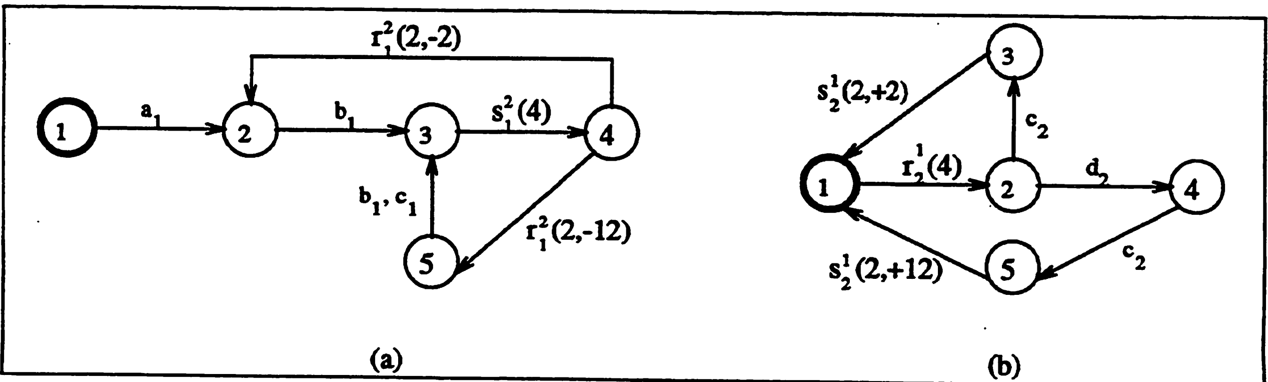


Figure 15: Specifications of the local controllers (obtained at step 5 of *Synt_Loc_Sup*). (a) $S_{2,1}$. (b) $S_{2,2}$.

Hybrid Dynamics: Continuous and Discrete Systems

Peter J. Ramadge *
Department of Electrical Engineering
Princeton University, Princeton, N.J., 08544
ramadge@ee.princeton.edu

April 1994

EXTENDED ABSTRACT

1 Introduction

This article is a summary of recent research on hybrid systems but it is by no means a comprehensive survey. It reports some recent work in the control community and touches on some related work in the computer aided verification community. Most of the material concentrates on two special examples which are presented and analyzed in detail.

Roughly, a hybrid system is a dynamical system consisting of coupled continuous and discrete subsystems. For our purposes one can think of a set of o.d.e.s coupled with a finite automaton. The o.d.e.s determine the local evolution of a set of continuous variables, the continuous variables induce state changes of the automaton and the state of the automaton determines the vector field of the o.d.e.. Two examples of switched flow control will be presented and analyzed later.

Hybrid systems arise in a variety of settings. Most notably when digital computers are used to control the flow of continuous variables, but also in other situations where the dynamics of continuous variables are determined by discrete actions. For example, in some flow models of manufacturing systems or in protocols with timing constraints.

The behavior of a hybrid system can be very complex. In special cases it is possible to reduce the continuous components to a 'higher-level' automaton model. In other cases the continuous dynamics, by the introduction of chaotic behavior, make the adoption of a statistical model more appropriate.

*Research partially supported by the National Science Foundation under grant ECS-9022634

The concept of hybrid systems is certainly not new. Witsenhausen [25] considered a class of hybrid-state continuous time systems in 1966. There is currently a renewed interest in this area due to the widespread use of digital computers in tasks requiring the regulation of continuous dynamics. For example, [23], [24], [13], and [11] deal with the issue of the quantization of continuous variables in a feedback loop; [14], [22], and [18] concern modeling frameworks for hybrid systems; Brockett investigates hybrid models for motion control in robotics [5]; and [20], [7] consider the dynamic behavior of a class of hybrid systems.

In the verification community there has also be recent interest in using hybrid models to develop tools for the automatic verification of so-called embedded computer systems: computers dedicated to the regulation of devices and their dynamics. For example, Alur and Dill [1], [2] develop a theory of timed automata. This framework is then extended in [3] to a general class of hybrid automata. The main issue in this work is the verification that a hybrid system exhibits a desired behavior.

A timed automaton is a hybrid system consisting of a set of simple integrators (clocks) coupled with a finite state automaton. Such systems can be used, for example, to model protocols with timing requirements and constraints. For timed automata, Alur and Dill [1] have introduced a temporal reduction of the continuous dynamics to a finite state automaton and proposed the use of this reduction to answer certain verification questions. Alur and others [3] have also examined the verification question in the case of more complex dynamics that might arise, for example, in the regulation of continuous variables: pressure, temperature, etc. It is shown that even for simple linear dynamics the verification questions can be undecidable.

Similar methods and observations have been employed to analyze the behavior of hybrid systems from a dynamical systems point of view. The paper [7] introduces two simple examples of hybrid systems that result from switched flow control. Using a reduction to a finite automaton one of these systems is shown to be generically periodic; the other system is shown to be chaotic. The chaotic nature of the dynamics suggests computational difficulty and may tie in with the un-decidability results from the verification literature.

In the remainder of the paper will be be concerned with the properties of the two examples from [7].

2 Switched Flow Systems

2.1 The Switched Arrival System

Consider a system consisting of N buffers, and one server. We refer to the contents of a buffer as 'work'; it will be convenient to think of work as a fluid, and a buffer as a tank. Work is removed from buffer i at a fixed rate $\rho_i > 0$ and the server delivers material to any selected buffer at unit rate. We assume that the system is closed, so that $\sum_{i=1}^{N} \rho_i = 1$. The location of the server may be selected using a feedback policy. Moving the server alters the topology of the flow, and hence permits control over the buffer levels. Figure 1 (a) shows the set-up with $N = 3$ and the server in location 1.

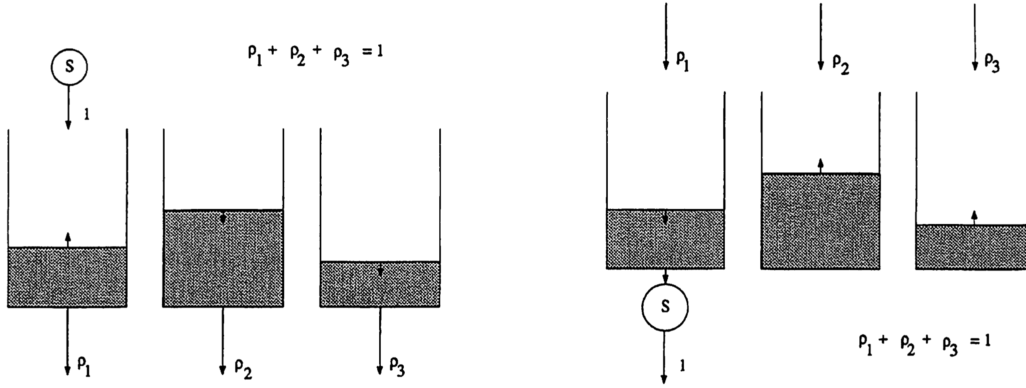


Figure 1: (a) The switched arrival system, (b) The switched server system.

In applications work can represent a continuous approximation to the discrete flow of parts in a manufacturing system, or jobs in a computer system, etc. Since each buffer acts as an integrator, the example can also be thought of as a switched set of simple o.d.e.s.

A interesting control policy is to assign a threshold to each buffer and instantaneously move the server to any buffer in which the level of work falls below the assigned threshold. In what follows we take all the thresholds equal to zero, and switch the arrival server each time a buffer empties. A more general analysis with both upper and lower thresholds can be found in [15].

Let $w_i(t)$ denote the amount of work in buffer i at time $t \geq 0$, and let $w(t) = (w_1(t), \ldots, w_N(t))$. At $t = 0$ we assume that $w_i(0) \geq 0$ with $\sum_{i=1}^{N} w_i(0) = 1$. We call $w(t)$ the *state of the buffers at time t*. If at time t the server is in location j, then the server will remain at j until a buffer empties. This event will occur after a time $\tau = \min_{i \neq j}\{w_i(t)/\rho_i\}$. For $t \leq s \leq t + \tau$ the buffer state is determined by the set of linear equations

$$w_i(s) = \begin{cases} w_i(t) - \rho_i(s-t), & \text{if } i \neq j; \\ w_i(t) + (1-\rho_i)(s-t), & \text{if } i = j. \end{cases} \tag{1}$$

At time $t + \tau$ one or more buffers are empty. If exactly one buffer is empty then its index is $k = \operatorname{argmin}_{i \neq j}\{w_i(t)/\rho_i\}$. In this case the arrival server is instantaneously moved to fill buffer k, and the above process repeats. If more than one buffer is empty we assume the system stops, so that $w(s) = w(t + \tau)$ for all $s \geq t + \tau$.

Let $\{t_n\}$ be the sequence of times when buffers empty, we refer to these as the *clearing times* of the system, and let $\tau_n = t_n - t_{n-1}$, $i \geq 1$. $\{\tau_n\}$ is the sequence of *inter-event times*, i.e., the times between the emptying of buffers. The case of particular interest is when $\sum_{n=1}^{\infty} \tau_n = \infty$, i.e., there are an infinite number of events and $\lim_{n\to\infty} t_n = \infty$. We verify in due course that this situation is in fact 'typical'.

Since the rate of work being processed is equal to the rate of work arriving, the total work in the buffer system is constant. The buffer state thus evolves in continuous time in the region of $\mathbb{R}^N$ defined by the intersection of the hyper-plane $\sum_i w_i = 1$ with the regions $w_i \geq 0$, $i = 1, \ldots, N$.

By sampling the trajectories at the clearing times we obtain an equivalent discrete-time

model. [1] Let $x_k(n) = w_k(t_n)$, and $x(n) = (x_1(n), \dots, x_N(n))$. At each clearing time the index of the empty buffer determines the new location of the server, and once this is known the value of the state until the next clearing time is determined by (1). Thus the sampled sequence $\{x(n): n \geq 0\}$, completely specifies the buffer trajectory. For simplicity we assume that 0 is a clearing time, i.e., that the initial condition has one buffer empty. This ensures that the initial condition is the first element in the sampled sequence.

The sequence $\{x(n)\}$ lies in the set $X = \{x\colon \sum_i x_i = 1, x_i \geq 0, \text{ and for some } j, x_j = 0\}$. If $G_j\colon X \to X$ denotes the map that describes the transformation of X that results by placing the server in location j until a buffer empties, then

$$G_j(x) = x + (\min_{k \neq j} \{x_k/\rho_k\})(1_j - \rho) \tag{2}$$

where ρ is the vector of the ρ_i, and 1_j is the vector of zeros except for a 1 in the jth position. Note that if $x \in X$ with $x_k = 0$ and $k \neq j$, then $G_j(x) = x$. So G_j only modifies $x \in X$ when x_j is the only zero element of x. The transition function of the sampled system, $G\colon X \to X$, is then given by $G(x) = G_{q(x)}(x)$ where $q(x) = \text{argmin}_i\{x_i\}$. If $q(x)$ is not uniquely defined, any of the indices minimizing x_i can be selected. This corresponds to the unlikely event that two or more buffers empty at exactly the same time. The state so reached is a fixed point of the transition function, i.e., $G(x) = x$. The set of initial conditions that give rise to such trajectories is easily shown to be of zero Borel measure.

To illustrate what is happening under G consider the system with $N = 3$. In this case the state space X can be visualized geometrically as the equilateral triangle in $\mathbb{R}^3$ with 'edges' X_1, X_2, X_3, where $X_i = \{x\colon \sum_1^3 x_i = 1, x_j > 0 \text{ for } j \neq i, x_i = 0\}$; and 'vertices' $v_1 = (1,0,0)$, $v_2 = (0,1,0)$, and $v_3 = (0,0,1)$. The vertices represent states where two buffers have emptied simultaneously; these are the fixed points of G.

Let $x \in X_1$, i.e., $x = (0, x_2, x_3)$, where $x_2 > 0, x_3 > 0$. Since $x_1 = 0$, the server starts filling buffer 1 at rate 1, while buffers 2 and 3 empty at rates $\rho_2 > 0$ and $\rho_3 > 0$. If $\frac{x_2}{\rho_2} < \frac{x_3}{\rho_3}$, then buffer 2 empties first and $G(x) \in X_2$. If $\frac{x_2}{\rho_2} > \frac{x_3}{\rho_3}$, then buffer 3 empties first and $G(x) \in X_3$. When $\frac{x_2}{\rho_2} = \frac{x_3}{\rho_3}$, both buffers empty simultaneously, and $G(x) = v_1$. This last event occurs when $x = P_1$, where P_1 is the point $(0, \frac{\rho_2}{\rho_2+\rho_3}, \frac{\rho_3}{\rho_2+\rho_3})$. The state transitions under G are illustrated in Figure 2 (a).

2.2 The Switched Server System

The second example consists of N buffers, with work arriving to buffer i at a constant rate of $\rho_i > 0$, and a server that removes work from any selected buffer at unit rate. As in the previous example the location of the server can be chosen using a feedback policy. We assume that the system is closed, so that $\sum_i \rho_i = 1$. Figure 1 (b) shows the set-up when $N = 3$ and the server is in location 1. [2]

An interesting policy can be formulated as follows. The server remains in its current

[1] This is analogous to forming a Poincaré map.

[2] This is a closed version of the model of [19].

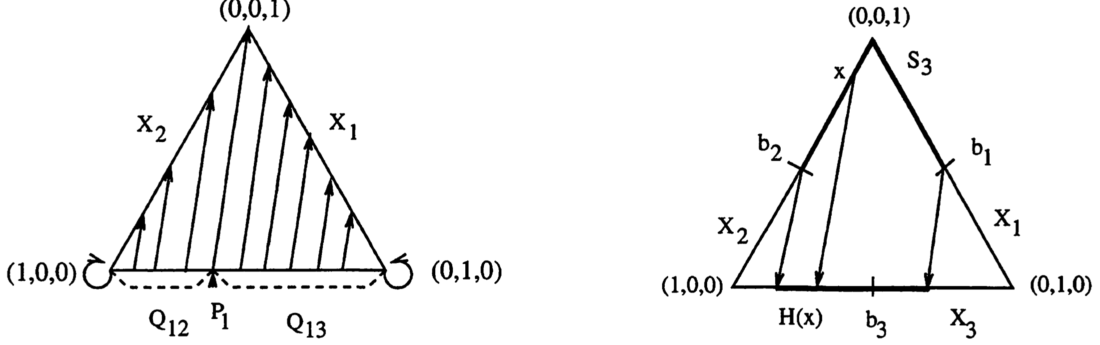


Figure 2: (a) Transitions under G, and (b) Transitions under H.

location until the associated buffer empties. Then the server instantaneously switches to a new buffer determined by a deterministic function of the current buffer state.

To determine the system equations we proceed as follows. First, for $t \geq 0$, let $w_i(t)$ denote the amount of work in buffer i, and let $w(t) = (w_1(t), \ldots, w_N(t))$. If the server is in location j at time t, then the server remains there until the event "buffer j empty" occurs. This takes a time $\tau = w_j(t)/(1 - \rho_j)$. For $t \leq s \leq t + \tau$, the buffer state changes linearly:

$$w_i(s) = \begin{cases} w_i(t) + \rho_i(s - t), & \text{if } i \neq j; \\ w_i(t) - (1 - \rho_i)(s - t), & \text{if } i = j. \end{cases} \tag{3}$$

When the buffer empties at time $t + \tau$ the server instantaneously switches to the buffer determined by a given feedback rule, $S: \mathbb{R}^N \to \{1, \ldots, N\}$, and then the process repeats.

As with the switched arrival system, we let $\{t_n\}$ and $\{\tau_n\}$ denote, respectively, the sequences of clearing times and inter-event times. Once more, the interesting case is when $\sum_1^\infty \tau_n = \infty$. The total work in the buffer system is constant, and the buffer state evolves in continuous time according to (3) in the region of $\mathbb{R}^N$ defined by the intersection of the hypersurface $\sum_i w_i = 1$ with the regions $w_i \geq 0$, $i = 1, \ldots, N$.

By sampling the system trajectories at the clearing times we obtain an equivalent discrete-time model. Let $x_k(n) = w_k(t_n)$ and $x(n) = (x_1(n), \ldots, x_N(n))$. Reasoning as in the previous case, this sequence completely determines the buffer state for all $t \geq 0$. For convenience assume that time 0 is a clearing time. The sequence $\{x(n)\}$ lies in the region X of $\mathbb{R}^N$ defined in the previous subsection. Let $H_j : X \to X$, be the map describing the transformation of X due to clearing buffer j. Then H_j is linear with

$$H_j(x)_i = \begin{cases} x_i + \frac{\rho_i}{1-\rho_j} x_j, & \text{if } i \neq j; \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

The transition function $H: X \to X$ of the sampled system is defined by $H(x) = H_{S(x)}(x)$, where $S: X \to X$ is the switching function that determines the next buffer to serve at state x. Clearly the $S(x)^{th}$ component of $H(x)$ will be zero.

It will not be necessary to specify an exact form for the switching function S. However, we assume that when switching function S is applied to a buffer state x, the buffer selected for service is nonempty.

To illustrate what is happening under H consider the system with $N = 3$, and the switching function $S(x) = \text{argmax}\{x_i\}$. In this case the state space is the one dimensional manifold X discussed in the previous subsection. The map S partitions X into three regions $S_i = S^{-1}(i)$, $i = 1, 2, 3$, each of which is a connected component of X. Three boundary points b_1, b_2, b_3 separate these regions. Let $x \in S_3$, with $x \in X_2$. So $x = (x_1, 0, x_3)$ with $x_3 > x_1$. Then buffer 3 is selected to be cleared and the next state is $H_3(x) \in X_3$. Clearly H maps S_3 into X_3. The state transition is illustrated in Figure 2 (b).

Intuitively, the switched arrival and switched server systems are "inverses" of each other. See [7] for details.

3 Analysis of the Switched Arrival System

We restrict our analysis here to the case $N = 3$. It can be shown that the closed loop hybrid system exhibits characteristics of chaos [12, page 50]: it has sensitive dependence on initial conditions, is topologically transitive, and its periodic orbits are dense in the state space. It is clear that there are three fixed points and it is easy to show that each of these is unstable. Similarly, it is possible to show that each periodic orbit is unstable. Thus although the state trajectory remains bounded it is highly unlikely that it will settle into a periodic pattern.

An alternative to direct analysis of the state trajectory is to model the initial condition as a random variable and examine the corresponding sequence of induced measures on the state space. Assuming certain regularity conditions these measures have density functions, and we can study the evolution of these functions with time.

3.1 Statistical analysis of the switched arrival system

The theory of the statistical analysis of deterministic functions on the unit interval is fairly well developed. For convenience in appealing to these results we first recast our system as a map on the unit interval. Let $\phi_1 : X_1 \to (0, 1/3)$, $\phi_2 : X_2 \to (1/3, 2/3)$, and $\phi_3 : X_3 \to (2/3, 1)$, where

$$\phi_1 : (0, x_2, x_3) \mapsto \frac{x_3}{3}; \phi_2 : (x_1, 0, x_3) \mapsto \frac{x_1 + 1}{3}; \phi_3 : (x_1, x_2, 0) \mapsto \frac{x_2 + 2}{3}$$

We map $(0, 1, 0)$ to 0, $(0, 0, 1)$ to $1/3$, and $(1, 0, 0)$ to $2/3$. This defines a one-to-one and onto map $\phi : X \to [0, 1)$. Geometrically, this parameterization of X can be thought of as "cutting" the triangle X at the point $(0, 1, 0)$ and "unfolding it" onto the unit interval.

We then bring in the induced transition function $g : [0, 1] \to [0, 1]$ defined on $[0, 1)$ by $g = \phi \circ G \circ \phi^{-1}$, and at the point 1 by $g(1) = 0$. Since the changes of coordinates ϕ_1, ϕ_2, ϕ_3

are affine, g is also piecewise linear. Specifically, for $z \in [0,1]$

$$g(z) = \begin{cases} 1 - \frac{\rho_2+\rho_3}{\rho_3} z & z \in [0, p_1) \\ \frac{2}{3} - \frac{\rho_2+\rho_3}{\rho_2}(z - p_1) & z \in [p_1, \frac{1}{3}) \\ \frac{1}{3} - \frac{\rho_1+\rho_3}{\rho_1}(z - \frac{1}{3}) & z \in [\frac{1}{3}, p_2) \\ 1 - \frac{\rho_1+\rho_3}{\rho_3}(z - p_2) & z \in [p_2, \frac{2}{3}) \\ \frac{2}{3} - \frac{\rho_1+\rho_2}{\rho_2}(z - \frac{2}{3}) & z \in [\frac{2}{3}, p_3) \\ \frac{1}{3} - \frac{\rho_1+\rho_2}{\rho_1}(z - p_3) & z \in [p_3, 1]. \end{cases}$$

where $p_1 = \frac{1}{3} \cdot \frac{\rho_3}{\rho_2+\rho_3}$, $p_2 = \frac{1}{3} + \frac{1}{3} \cdot \frac{\rho_1}{\rho_1+\rho_3}$, and $p_3 = \frac{2}{3} + \frac{1}{3} \cdot \frac{\rho_2}{\rho_1+\rho_2}$. These are the preimages under g of of $\frac{2}{3}, 0$, and $\frac{1}{3}$, respectively.

Our underlying measure space is $(I, \mathcal{A}, m)$ with $I = [0,1]$ and Lebesgue measure m. Let L_1 denote the family of integrable functions on $[0,1]$, and $\mathcal{D} \subset L_1$ denote the family of density functions. The transformation g is nonsingular with respect to m, i.e., for each $A \in \mathcal{A}, m(A) = 0$ implies $m(g^{-1}(A)) = 0$. Hence g induces a Markov operator $P_g: L_1 \to L_1$, called the *Frobenius Perron operator* of g [16, p.37]. If the initial condition $z(0)$ for the system $([0,1], g)$ is a random variable on the probability space $(I, \mathcal{A}, m)$ with density function f_0, then the next state $z(1) = g(z(0))$ is a random variable with a density function $f_1 = P_g(f_0)$.

The fixed points of P_g in D are called *stationary densities.* These represent statistically stationary regimes of operation. The map g is *statistically stable* if there exists a stationary density f^* such that for any density $f \in D$, $\lim_{n\to\infty} \|P_g^n(f) - f^*\|_1 = 0$. In this case, regardless of the initial density the state will asymptotically be distributed with density f^*. In this sense the system has a unique "steady state".

Our main result for the sampled switched arrival system is the following [7].

Theorem 3.1 *The map g representing the sampled switched arrival system on $[0,1]$ is statistically stable and the unique stationary density is the piecewise constant function*

$$f^*(z) = \begin{cases} \frac{3}{2} \frac{\rho_1(1-\rho_1)}{\rho_1\rho_2+\rho_1\rho_3+\rho_2\rho_3}, & \textit{if } z \in [0, 1/3); \\ \frac{3}{2} \frac{\rho_2(1-\rho_2)}{\rho_1\rho_2+\rho_1\rho_3+\rho_2\rho_3}, & \textit{if } z \in [1/3, 2/3); \\ \frac{3}{2} \frac{\rho_3(1-\rho_3)}{\rho_1\rho_2+\rho_1\rho_3+\rho_2\rho_3}, & \textit{if } z \in [2/3, 1]. \end{cases}$$

In addition, g is measure preserving and ergodic on the measure space $(I, \mathcal{A}, \mu^*)$, where μ^* is the measure induced by f^*.

The proof of this theorem is contained in [7]. However, it is easy to give a partial justification of the result. Using standard methods [16, p.38], we calculate the Frobenius-Perron operator of g to be

$$P_g(f)(z) = \begin{cases} \frac{\rho_1}{\rho_1+\rho_3} f(p_2 - \frac{z\rho_1}{\rho_1+\rho_3}) + \frac{\rho_1}{\rho_1+\rho_2} f(1 - \frac{z\rho_1}{\rho_1+\rho_2}) & z \in [0, \frac{1}{3}) \\ \frac{\rho_2}{\rho_2+\rho_3} f(\frac{1}{3} - \frac{(z-\frac{1}{3})\rho_2}{\rho_2+\rho_3}) + \frac{\rho_2}{\rho_1+\rho_2} f(p_3 - \frac{(z-\frac{1}{3})\rho_2}{\rho_1+\rho_2}) & z \in [\frac{1}{3}, \frac{2}{3}) \\ \frac{\rho_3}{\rho_2+\rho_3} f(p_1 - \frac{(z-\frac{2}{3})\rho_3}{\rho_2+\rho_3}) + \frac{\rho_3}{\rho_1+\rho_3} f(\frac{2}{3} - \frac{(z-\frac{2}{3})\rho_3}{\rho_1+\rho_3}) & z \in [\frac{2}{3}, 1]. \end{cases}$$

From the expression for P_g we can make some simple observations. First, let $\mathcal{D}_c$ denote the set of density functions that are constant on each of the subintervals $[0, 1/2)$, $[1/3, 2/3)$ and $[2/3, 1]$, i.e., if $f \in \mathcal{D}_c$, then f has the form $f(z) = \alpha_1 1_{[0,1/3)}(z) + \alpha_2 1_{[1/3,2/3)}(z) + \alpha_3 1_{[2/3,1]}(z)$. Then $\mathcal{D}_c$ is invariant under P_g, i.e., for each $f \in \mathcal{D}_c$, $P_G(f) \in \mathcal{D}_c$. Now any $f \in \mathcal{D}_c$ can be represented as a triple $\alpha = (\alpha_1, \alpha_2, \alpha_3) \in [0, 1]^3$, with $\alpha_1 + \alpha_2 + \alpha_3 = 3$, and the action of P_g on f can be represented as a matrix product $\mathcal{P}\alpha$ where $\mathcal{P}$ is the 3×3 matrix

$$\mathcal{P} = \begin{bmatrix} 0 & \frac{\rho_1}{\rho_1+\rho_3} & \frac{\rho_1}{\rho_1+\rho_2} \\ \frac{\rho_2}{\rho_2+\rho_3} & 0 & \frac{\rho_2}{\rho_1+\rho_2} \\ \frac{\rho_3}{\rho_2+\rho_3} & \frac{\rho_3}{\rho_1+\rho_3} & 0 \end{bmatrix}.$$

A simple calculation shows that $\mathcal{P}$ is the transpose of an irreducible aperiodic Markov matrix. Hence by the standard Frobenius-Perron theorem for nonnegative matrices we know that 1 is a simple eigenvalue of $\mathcal{P}$, and that all other eigenvalues have absolute value strictly less than 1. Moreover, for the eigenvalue 1 there exists a unique strictly positive eigenvector α^*, with $\sum_i \alpha_i^* = 3$, and for any $\alpha \in \mathbb{R}^3$, with $\sum_i \alpha_i = 3$, $\mathcal{P}^n\alpha \to \alpha^*$ as $n \to \infty$.

Let the piecewise constant density function corresponding to α^* be denoted by f^*. Clearly f^* is a stationary density of P_g. Moreover, for any piecewise constant density $f \in \mathcal{D}_c$, $\lim_{n\to\infty} \|P_g^n(f) - f^*\| = 0$. This suggests that f^* is a potential globally attractive stationary density. To show statistical stability, however, we have to show that *every* $f \in \mathcal{D}$ converges to f^* (in L_1) under the iterations of P_g. This requires a more detailed argument.

The fact that g is ergodic and measure preserving with respect to f^* allows us to appeal to the Birkhoff Ergodic Theorem to equate (almost surely) sample path averages with expectations. Specifically, for any bounded measurable function, h, on I, and for almost all initial conditions $z(0) \in I$

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} h(g^i(z(0))) = \int_0^1 h(z) f^*(z) dz$$

i.e., time averages of $h(z(n))$ are equal a.s. to the expected value of h with respect to f^*.

For example, let $\tau : X \to \mathbb{R}^+$ with $\tau(x) = \min_{i \neq j} x_i/\rho_i$, where j is any index for which $x_j = 0$. For $x \in X$, $\tau(x)$ is the service time at the buffer state x, i.e., the time until the next buffer empties when the buffer state is x. For a buffer trajectory $\{x(n)\}$, $\tau(x(n))$ is the $(n+1)^{\text{th}}$ inter-event time. Using the coordinate change ϕ, we can obtain an equivalent measurable function $\hat{\tau} = \tau(\phi^{-1})$ which gives the service time as a function of the state $z \in I$. The average inter-event time for initial state $z(0)$ is $\bar{\tau} = \lim_{n\to\infty} \frac{1}{n}\sum_{i=0}^{n} \hat{\tau}(g^i(z(0)))$, and by the Birkhoff Ergodic Theorem this a.s. equals $\int_0^1 \hat{\tau}(z) f^*(z) dz$. Evaluation of this integral yields $\bar{\tau} = 1/(4d)$, where $d = \rho_1\rho_2 + \rho_1\rho_3 + \rho_2\rho_3$. Since $\bar{\tau} > 0$, the inter-event times, $\{\tau_i\}$, sum to infinity almost surely.

When the system is stationary, e.g., the initial state is a random variable with density f^*, the inter-event times have fixed first order statistics. For example, if $\rho_1 \geq \rho_2 \geq \rho_3$, then the density f_τ^* of the service times is given by

$$f_\tau^*(t) = \begin{cases} \sum_{i=1}^{3} \frac{\rho_i(1-\rho_i)^2}{2d} & \text{if } t \in \left[0, \frac{1}{1-\rho_3}\right]; \\ \sum_{i=1}^{2} \frac{\rho_i(1-\rho_i)^2}{2d} & \text{if } t \in \left(\frac{1}{1-\rho_3}, \frac{1}{1-\rho_2}\right]; \\ \frac{\rho_1(1-\rho_1)^2}{2d} & \text{if } t \in \left(\frac{1}{1-\rho_2}, \frac{1}{1-\rho_1}\right] \end{cases}$$

where $d = \rho_1\rho_2 + \rho_1\rho_3 + \rho_2\rho_3$.

4 Analysis of the Switched Server System

We now consider the sampled switched server system with attention restricted to the case $N = 3$. For $N = 3$ the state space is the one dimensional manifold X defined in Section 2.

We use the topology on X induced by the Euclidean topology of R^3. For $B \subset X$, $\overline{B}$ denotes the closure of B and ∂B denotes the set of boundary points of B. We let μ denote the one-dimensional Hausdorff measure on X. The μ-measure of a connected component in X corresponds to its path length.

The parameter space for the system is $\Theta = \{(\rho_1, \rho_2, \rho_3) | \sum \rho_i = 1, \rho_i > 0, i = 1, 2, 3\}$. On Θ we use the topology induced by the Euclidean norm in $\mathbb{R}^3$.

The switching function $S: X \to \{1, 2, 3\}$ partitions X into switching sets, $\{S_i\}_1^3$, with $S_i = S^{-1}(i)$. We assume:

[S1] Each switching set S_i has a finite number of connected components.

[S2] There exists $\alpha > 0$ such that for all $x \in X$, $x_{S(x)} \geq \alpha$.

Let $\partial S = \cup_i \partial S_i$. We refer to points in ∂S as the *switching points* or *boundary points.* Assumption [S1] ensures that ∂S is a finite set. Indeed, [S1] and the topology of X imply that each switching point is a boundary point of exactly two switching sets. So the number of switching points is equal to the total number of connected components of the switching sets.

Assumption [S2] guarantees that a minimum amount of work is present in the buffer selected for service. This is equivalent to the Clear-A-Fraction property introduced in [19]. It guarantees that the inter-event times sum to infinity.

Let π_0 denote the equivalence relation on X induced by S, i.e., the equivalences classes of π_0 are the switching sets S_i, and let π be the equivalence relation on X with $x \equiv y (\text{mod } \pi)$ iff for all $n \geq 0$, $S(H^n(x)) = S(H^n(y))$, i.e., iff x and y produce the same control sequence. It is easy to see that π is a congruence on (X, H), i.e., that $x \equiv y (\text{mod } \pi)$ implies that $H(x) \equiv H(y) (\text{mod } \pi)$. If $S(x)$ is thought of as an 'observation' of the state $x \in X$, then π is the *observability congruence* of the system (X, H, S).

Let $\bar{X}$ denote the set of equivalence classes of π, and let π also denote the canonical projection $\pi: X \to \bar{X}$. Then there exists a unique map $\bar{H}: \bar{X} \to \bar{X}$ such that $\pi H = \bar{H}\pi$. Since π is finer than π_0, and $x \equiv y (\text{mod } \pi_0)$ iff $S(x) = S(y)$, it is clear that there exists a map $\bar{S}: \bar{X} \to \{1, \ldots, N\}$ such that $S = \bar{S}\pi$. The above observations are summarized in the commutative diagram:

$$\begin{array}{ccccc} X & \xrightarrow{H} & X & \xrightarrow{S} & \{1, \ldots, N\} \\ \pi \downarrow & & \downarrow \pi & \bar{S} \nearrow & \\ \bar{X} & \xrightarrow{\bar{H}} & \bar{X} & & \end{array}$$

The situation of interest is when the quotient system $(\bar{X}, \bar{H}, \bar{S})$ has a finite number of states, i.e., π has finite index. In this case the control structure of the switched server system will be reduced to a finite automaton, i.e., the switching sequence and hence the control policy will be completely determined by a deterministic finite state system. As a result the control policy will be eventually periodic.

Finally, for each switching function S and for all n, $H^{n+1}(X) \subseteq H^n(X)$. So $X \supseteq \overline{H(X)} \supseteq \overline{H^2(X)} \supseteq \cdots$. The limit set is the forward attractor $\Lambda = \cap_{n=1}^{\infty} \overline{H^n(X)}$. This is the set of limit points of all possible trajectories.

The main result for the sampled switched server system is the following [7].

Theorem 4.1 *For each fixed $\rho \in \Theta$ there is a set $\Gamma_\rho \subset X$ of measure zero, such that for all switching functions S having switching points outside of Γ_ρ, the following hold:*

1. *The observability congruence π has finite index.*
2. *Λ contains at most $2|\partial S|$ possible periodic cycles.*
3. *All buffer trajectories converge uniformly exponentially to periodic orbits.*
4. *Items 1-3 continue to hold for sufficiently small variations of the switching function S (meaning small changes in the switching points).*

In addition, there exists a set $\Gamma \subset X$ of measure zero such that for each switching function having switching points outside Γ there is an open dense set in the parameter space for which items 1-3 hold.

Roughly, the theorem says that for almost all switching functions S the control policy of the sampled switched server system is determined by a finite automaton and is thus eventually periodic. This in turn implies that the buffer trajectory is asymptotically periodic. Moreover, the system retains this qualitative behavior for sufficiently small changes in the system or controller parameters. So these characteristics are structurally stable.

Notice that the quotient automaton gives all important information regarding the controller dynamics. It displays both the transient as well as the steady-state controller behavior. Given the steady-state controller behavior it is a simple matter to actually compute the asymptotic periodic orbit for the buffer state.

There remains the problem of computing the finite automaton $(\bar{X}, \bar{H}, \bar{S})$. It is easy, in this example, to formulate an algorithm by which this may be done, and this algorithm indicates that the automaton $(\bar{X}, \bar{H}, \bar{S})$ is structurally stable with respect to variations of the switching function S, and the parameters ρ. However, since the algorithm makes use of the (expansive) inverse map G, it would appear to have undesirable characteristics.

5 Conclusions

There are several technical difficulties involved in extending our results to higher dimensional systems. The available results on the statistical stability of higher dimensional systems

are inapplicable, and a complete analysis of discontinuous piecewise contractions in higher dimensions is an open problem. Some issues that arise in proving statistical stability have been examined in a general setting in [21] following the method of [17]. In addition, the state transition function of the sampled N buffer switched server system is an example of a Markov map [4], and the statistical stability of such maps is investigated in [8]. Work on contractive systems has specifically concerned the N buffer system [10], and discrete time systems on the unit interval where a controller selects among a finite number of contractive transition maps [9]. The method of analysis used here for the switched server system does not immediately extend to the higher dimensional case. Aside from the results of [10], this remains an open problem.

References

[1] R. Alur and D. L. Dill, "Automata for Modeling Real-Time Systems," *Proceedings of the 17th International Colloquium: Automata, Languages and Programming*, pp. 322-335, Coventry, U.K., July 16-20, 1990.

[2] R. Alur and D. L. Dill, "The theory of timed automata," *Proceedings of the REX Workshop, Real-Time: Theory in Practice*, pp. 45-73, Mook, The Netherlands, June 3-7, 1991.

[3] R. Alur, C. Courcoubetis, T. A. Henzinger and Pei-Hsin Ho, Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems, *Proceedings of the Workshop on Theory of Hybrid Systems*, 1993.

[4] A. Boyarsky and M. Scarowsky, "On a Class of Transformations Which Have Unique Absolutely Continuous Invariant Measures," *Transactions of the American Mathematical Society,* Vol. 255, pp. 243-262, November 1979.

[5] "Hybrid models of motion control systems," in *Perspectives in Control,* H. L. Trentelman and J. C. Willems, Eds, Birkhauser, 1993.

[6] S. Buss, C. H. Papadimitriou, J. Tsitsiklis, "On the predictability of coupled automata: an allegory about chaos," *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,* pp. 788–793, Oct. 1990.

[7] C. J. Chase, L. J. Serrano, and P. J. Ramadge, "Periodicity and chaos from switched flow systems: contrasting examples of the discrete control of continuous systems," *IEEE Transactions on Automatic Control,* Jan. 1993.

[8] C. J. Chase, "Analysis of Dynamics in Continuous Variable Systems with a Scheduled Controller," Ph.D. dissertation, Department of Electrical Engineering, Princeton University, January 1992.

[9] C. J. Chase and P. J. Ramadge, "Predictability of a Class of Supervised One-dimensional Systems," in *Proc. 5th IEEE Inter. Symposium on Intelligent Control*, Philadelphia, PA, Sept. 1990.

[10] C. J. Chase and P. J. Ramadge, "Dynamics of a Switched N Buffer System," in *Proc. 28th Allerton Conf. on Communication, Control, and Computing*, Champaign, IL, Oct. 1990.

[11] D. F. Delchamps "The Stabilizing a Linear System with Quantized State Feedback," *IEEE Transactions on Automatic Control*, vol. 35, no. 8, pp. 916-924, August 1990.

[12] R. L. Devaney *Chaotic Dynamical Systems.* Addison-Wesley, New York, N.Y., 1989.

[13] R. M. Gray, W. Chou, and P.-W. Wong, "Quantization noise in single-loop sigma-delta modulation with sinusoidal inputs," *IEEE Trans. Commun.*, vol. COM-37, pp. 956-968, 1989.

[14] A. Göllü and P. Varaiya, "Hybrid dynamical systems," *Proceedings of the 28th Conference on Decision and Control,* Tampa, Fl., pp. 2708–2712, Dec. 1989.

[15] C. Horn and P. J. Ramadge, "Dynamics of switched arrival systems with thresholds," Proceedings of the 32nd IEEE Conference on Decision and Control, San Antonio, Texas, Dec., 1993.

[16] A. Lasota and M. C. Mackey, *Probabilistic Properties of Deterministic Systems.* Cambridge University Press, New York, N.Y., 1985.

[17] T.-Y. Li and J. A. Yorke, "Ergodic Transformations from an Interval into Itself," *Transactions of the American Mathematical Society,* Vol. 235, pp. 183-192, 1978.

[18] P. Peleties and R. Decarlo, "A modeling strategy for hybrid systems based on event structures," Preprint: School of Electrical Engineering, Purdue University, W. Lafayette, IN 47907.

[19] J. R. Perkins and P. R. Kumar, "Stable, Distributed, Real-Time, Scheduling of Flexible Manufacturing/Assembly/Disassembly Systems," *IEEE Trans on Auto. Control,* vol. 34 (2), Feb. 1989, pp. 139–148.

[20] P. J. Ramadge, "On the Periodicity of Symbolic Observations of Piecewise Smooth Discrete-Time Systems," *IEEE Trans on Auto. Control,* vol. 35 (7), July 1990.

[21] L. J. Serrano, "The Effects of Time Sampling and Quantization in the Discrete Control of Continuous Systems," Ph. D. Thesis, Princeton University, Oct. 1990.

[22] J. A. Stiver and P. J. Antsaklis, "A novel discrete event system approach to modeling and analysis of hybrid control systems," Control Systems Technical Report #71, Dept. of Electrical Eng., University of Notre Dame, Notre Dame IN 46556, June 1991.

[23] T. Ushio and C. S. Hsu, "Simple example of a digital control system with chaotic rounding errors," *Int. J. Control,* 45 (1), pp. 17–31, 1987.

[24] T. Ushio and C. S. Hsu, "Chaotic rounding error in digital control systems," *IEEE Transactions on Circuits and Systems,* 34 (2), pp. 133–139, Feb. 1987.

[25] H. S. Witsenhausen, "A class of hybrid-state continuous-time dynamic systems," *IEEE Transactions on Automatic Control,* vol. AC-11, no. 2, April 1966.

Control of Discrete-Event Systems Under State Fairness Assumptions (Extended Abstract)*

J.G. Thistle and R.P. Malhamé
Section automatique
Département de génie électrique et de génie informatique
École Polytechnique de Montréal
C.P. 6079, Succ. Centre-ville
Montréal, (Québec)
Canada H3C 3A7

Abstract

This article studies the deadlock-free control of finite automata subject to specifications in the form of Rabin acceptance conditions. The automata are assumed to satisfy a *state fairness* condition, whereby any transition that is infinitely often enabled (by both the underlying dynamics and the control mechanism) must eventually occur. The problem of computing the automaton's *controllability subset* – the set of states from which it can be controlled to satisfy its acceptance condition – is solved through a fixpoint characterization of this state subset. The state fairness condition simplifies the fixpoint characterization and allows the controllability subset to be computed in polynomial time. The problem represents a modified version of Church's problem and the emptiness problem for automata on infinite trees, and has potential applications to the verification and synthesis of reactive systems and to supervisory control.

1 Introduction

This article relates to the extension of *supervisory control theory* [RW89] to the setting of infinite-string formal languages and the associated automata [Ram89]. The infinitary extension allows the examination of liveness issues within the context of the theory [TW94a, TW94b]. Such questions appear to be useful, for example, in the formulation of real-time supervisory control problems [WW93].

Earlier articles identify a basic control synthesis problem that amounts to one of controlling a finite automaton to the satisfaction of a specification in the form of a Rabin acceptance

*Research supported by a grant from the Fonds pour la formation de chercheurs et l'aide à la recherche of the province of Quebec, Canada, Bell-Northern Research Ltd. and the Natural Sciences and Engineering Research Council of Canada under the Action concertée sur les méthodes mathématiques pour la synthèse de systèmes informatiques.

condition [TW94a]. In its basic form, this problem is equivalent to Church's automaton synthesis problem and the emptiness problem for automata on infinite trees, which represent theoretical paradigms for the synthesis and verification of concurrent and reactive systems [EJ88, PR89]. In [Thi], results on this basic problem are extended to admit a second Rabin acceptance condition representing a liveness condition assumed to be satisfied by the automaton irrespective of control action.

The problem considered in [TW94a, Thi] is that of computing an automaton's *controllability subset* – the set of states from which it can be controlled to the satisfaction of its specification. The solutions proposed are polynomial in the size of the automaton's state set and exponential in the number of state subset pairs in the Rabin acceptance condition. These can be considered essentially optimal in the sense that the problems are NP-complete. In the present article we give a fixpoint characterization of a controllability subset that allows a polynomial-time computation. It applies in the case of automata that satisfy *state fairness* conditions. Such a condition states that any transition that is infinitely often enabled by the dynamics of the automaton (and, in our case, simultaneously by the control mechanism) will eventually occur.

Section 2 defines Rabin automata and their controllability subsets. In section 3, a fixpoint characterization of the controllability subset is given. Related work is discussed in section 4. Fixpoint preliminaries are presented in an appendix.

2 Controllability subsets

We first establish some formal language notation. A *string* is any sequence of symbols of some (finite) alphabet Σ. The set of all finite strings over Σ, including the *empty string*, 1 (having length zero), is denoted by Σ^*. The set of all (countably) infinite strings – or *ω-strings* – is denoted by Σ^ω. A string $k \in \Sigma^*$ is a *prefix* of a string $w \in \Sigma^* \cup \Sigma^\omega$ if it is an initial subsequence of w; in this case we write $k \leq w$, or $k < w$ if k is a strict subsequence of w.

A (deterministic) *Rabin automaton* is a 5-tuple

$$\mathcal{A} = (\Sigma, X, \delta, x_0, \{(R_p, I_p) : p \in P\}),$$

where

- Σ is an *alphabet* of *event symbols*;
- X is a finite *state set*;
- $\delta : \Sigma \times X \longrightarrow X$ (partial function) is a *transition function*;
- $x_0 \in X$ is an *initial state*; and
- $\{(R_p, I_p) : p \in P\}$ is a family of pairs of state subsets indexed by elements of some index set P.

As usual, we extend the transition function to strings by defining

$$\begin{array}{rcl} \delta : \Sigma^* \times X & \longrightarrow & X \\ (1, x) & \mapsto & x \\ (k\sigma, x) & \mapsto & \begin{cases} \delta(\sigma, \delta(k, x)), & \text{if } \delta(\sigma, \delta(k, x)) \text{ is defined} \\ \text{undefined}, & \text{otherwise} \end{cases} \end{array}$$

where $k \in \Sigma^*$ is any finite string and $\sigma \in \Sigma$ is any symbol. We shall write $\delta(k, x)!$ to signify that $\delta(k, x)$ is defined.

We shall interpret automata as *generators*, rather than acceptors, of strings. A finite string $k \in \Sigma^*$ is said to be *generated* by $\mathcal{A}$ if $\delta(k, x_0)!$.

Let $\mathbb{C} \subseteq 2^\Sigma$ represent a set of allowable control actions in the following sense: if $\Gamma \in \mathbb{C}$ then at any point in the evolution of the system, the set of transitions that can occur may be restricted to those labelled by an element of Γ. (In other words, the set of symbols that may be "generated" at any given time can be restricted to those belonging to Γ.) A controller can be modelled as a map $f : \Sigma^* \longrightarrow \mathbb{C}$ taking finite sequences of (past) events into control actions.

The set of finite strings *generated by $\mathcal{A}$ under f* is the set of all $k \in \Sigma^*$ generated by $\mathcal{A}$ such that for any prefix $k'\sigma$ of k, $\sigma \in f(k')$.

To discuss the infinite behaviour of $\mathcal{A}$ we assume without loss of generality that distinct transitions of $\mathcal{A}$ carry distinct event symbols; that is,

$$\begin{array}{l} \forall \sigma \in \Sigma, \forall x_1, x_2, x_3, x_4 \in X \ : \\ \qquad \delta(\sigma, x_1) = x_2 \ \& \ \delta(\sigma, x_3) = x_4 \Longrightarrow x_1 = x_3 \ \& \ x_2 = x_4 \end{array}$$

An infinite string $s \in \Sigma^\omega$ is said to be generated by $\mathcal{A}$ (under f) if

i. all finite prefixes of s are generated by $\mathcal{A}$ (under f); and if

ii. for any event symbol $\sigma \in \Sigma$ such that $k\sigma$ is generated by $\mathcal{A}$ (under f) for infinitely many finite prefixes k of s, σ appears infinitely often in s.

The second condition states effectively that any state transition that is infinitely often enabled according to the transition structure of $\mathcal{A}$ (and the control action of f) must occur infinitely often. Such a liveness assumption is known in the literature as a *state fairness* condition [CVW86].

We consider the problem of controlling the automaton $\mathcal{A}$, under this state fairness assumption, so that it generates only infinite strings that satisfy the Rabin acceptance condition.

To define satisfaction of the Rabin acceptance condition, let the set of states visited infinitely often during the generation of a string s – the *recurrence set* of s – be given by

$$\Omega_s := \{x \in X : |\{k < s : \delta(k, x_0) = x\}| = \omega\}$$

The string s is *accepted* if

$$\exists p \in P \ : \ \Omega_s \cap R_p \neq \emptyset \ \& \ \Omega_s \subseteq I_p$$

In other words, s is accepted if it traces out a path on the transition structure of $\mathcal{A}$ that begins at the initial state and intersects $R_p \subseteq X$ infinitely often and $I_p \subseteq X$ almost always, for some $p \in P$. In order to discuss acceptance of strings generated from states other than the initial one, let $\mathcal{A}_x$ be the automaton obtained from $\mathcal{A}$ by replacing the initial state x_0 with x.

The *controllability subset* $F^{\mathcal{A}} \subseteq X$ is defined as the set of all (initial) states $x \in X$ for which there exists a map $f : \Sigma^* \longrightarrow \mathbb{C}$ such that

i. every finite string generated by $\mathcal{A}_x$ under f extends to an infinite string generated by $\mathcal{A}_x$ under f; and

ii. every infinite string generated by $\mathcal{A}_x$ under f is accepted by $\mathcal{A}_x$.

The second clause of this definition requires that the controller restrict the infinite trajectories of the automaton to those that satisfy the acceptance condition; the first requires that the controlled system be deadlock-free.

3 Fixpoint characterization of the controllability subset

We shall characterize the controllability subset in terms of fixpoints of the *inverse dynamics operator* $\theta^{\mathcal{A}} : 2^X \longrightarrow 2^X$ that maps every state subset $X' \subseteq X$ to the set of states $x \in X$ such that for some $\Gamma \in \mathbb{C}$, there exists $\sigma \in \Gamma$ such that $\delta(\sigma, x)!$ and furthermore for all such $\sigma \in \Gamma$, $\delta(\sigma, x) \in X'$. In other words, $\theta^{\mathcal{A}}(X')$ is the set of states from which $\mathcal{A}$ can be forced to enter X' in a single transition.

Define in addition the operation $\mathcal{A}(\not\rightarrow X')$, for $X' \subseteq X$, which allows the controller to disable any transitions that terminate in X'. The operation thus replaces the set $\mathbb{C}$ of control actions with

$$\mathbb{C}' := \{\Gamma' \subseteq \Sigma : \ (\exists \Gamma \in \mathbb{C})[\Gamma \setminus \Sigma' \subseteq \Gamma' \subseteq \Gamma]\},$$

where $\Sigma' := \Sigma \setminus \{\sigma \in \Sigma : \ (\exists x \in X, x' \in X')[\delta(\sigma, x) = x']\}$.

Define the operator $\rho^{\mathcal{A}} : 2^X \longrightarrow 2^X$ so that

$$\rho^{\mathcal{A}}(X') = \nu X_0. \ \mu X_1. \ \theta^{\mathcal{A}(\not\rightarrow X_0)}(X_1 \cup X')$$

This fixpoint intuitively represents the set of states from which the automaton can be controlled to reach X' (under the state fairness assumption). The reader unfamiliar with the quantifier-style notation for fixpoints is referred to appendix A. Now define the following subset $C^{\mathcal{A}} \subseteq X$:

$$C^{\mathcal{A}} := \rho^{\mathcal{A}}\Big(\bigcup_{p \in P} C_p^{\mathcal{A}}\Big)$$

where $C_p^{\mathcal{A}} := \nu X_0. \ \mu X_1. \ [\theta^{\mathcal{A}(\not\rightarrow X_0)}(X_1 \cup (R_p \cap X_0)) \cap I_p]$ represents the largest subset $X_0 \subseteq I_p$ such that the system can be controlled so that (a) there exists a path from every $x \in X_0$ to R_p and (b) the system remains within X_0.

It is easy to see that $C^{\mathcal{A}}$ can be computed in polynomial time (see Theorem A.1). Our main result states that $C^{\mathcal{A}}$ represents a fixpoint characterization of the controllability subset $F^{\mathcal{A}}$. In order to simplify the proof, we bring in the following operations on automata:

restriction to a subset $X' \subseteq X$:

$\mathcal{A} \upharpoonright X' := (\Sigma, X, \delta', x_0, \{(R'_p, I'_p) : p \in P\})$, where

$$\delta'(\sigma, x') = \begin{cases} \text{undefined} & \text{if } \delta(\sigma, x') \text{ is undefined;} \\ \delta(\sigma, x') & \text{if } \delta(\sigma, x')! \;\&\; x' \in X'; \\ x' & \text{otherwise} \end{cases}$$
$$\& \; R'_p = R_p \cap X' \;\&\; I'_p = I_p \cap X', \; \forall p \in P$$

self-looping of a subset $X' \subseteq X$:

$\mathcal{A}(\hookrightarrow X') := (\Sigma, X, \delta', x_0, \{(R'_p, I'_p) : p \in P\})$, where

$$\delta'(\sigma, x') = \begin{cases} x' & \text{if } \delta(\sigma, x')! \;\&\; x' \in X'; \\ \delta(\sigma, x') & \text{otherwise} \end{cases}$$
$$\& \; R'_p = R_p \cup X' \;\&\; I'_p = I_p \cup X', \; \forall p \in P$$

We state without proof the following result on the effect of these operations on the controllability subset:

Proposition 3.1 Let $\mathcal{A} = (\Sigma, X, \delta, x_0, \{(R_p, I_p) : p \in P\})$ be a deterministic Rabin automaton equipped with a family of control actions $\mathbb{C} \subseteq 2^{\Sigma}$ as described above. Let $X' \subseteq X$. Then

$$\begin{array}{ll} (a) & F^{\mathcal{A} \upharpoonright X'} \subseteq F^{\mathcal{A}} \cap X' \\ (b) & X' \subseteq F^{\mathcal{A}} \Longrightarrow F^{\mathcal{A}(\hookrightarrow X')} = F^{\mathcal{A}} \end{array}$$

□

We also note that the fixpoint $C^{\mathcal{A}}$ shares the same properties:

Proposition 3.2 Let $\mathcal{A} = (\Sigma, X, \delta, x_0, \{(R_p, I_p) : p \in P\})$ be a deterministic Rabin automaton equipped with a family of control actions $\mathbb{C} \subseteq 2^{\Sigma}$ as described above. Let $X' \subseteq X$. Then

$$\begin{array}{ll} (a) & C^{\mathcal{A} \upharpoonright X'} \subseteq C^{\mathcal{A}} \cap X' \\ (b) & X' \subseteq C^{\mathcal{A}} \Longrightarrow C^{\mathcal{A}(\hookrightarrow X')} = C^{\mathcal{A}} \end{array}$$

□

We are now ready to state and prove the main result:

Proposition 3.3 Let $\mathcal{A}$ be a Rabin automaton equipped with a family of control actions as described above. Then $F^{\mathcal{A}} = C^{\mathcal{A}}$.

Proof: For the containment $F^{\mathcal{A}} \supseteq C^{\mathcal{A}}$, we construct a state feedback control $\phi : C^{\mathcal{A}} \longrightarrow \mathbb{C}$ based on the fixpoint definition of $C^{\mathcal{A}}$.

First, consider the definition of $C_p^{\mathcal{A}}$, $p \in P$. Note that by Theorem 1, $C_p^{\mathcal{A}} = \bigcup_{i=1}^{n} C_{p,i}^{\mathcal{A}}$, where

$$C_{p,0}^{\mathcal{A}} := \emptyset; \text{ and, for } i > 0,$$
$$C_{p,i+1}^{\mathcal{A}} := \theta^{\mathcal{A}(\not\to C_p^{\mathcal{A}})}(C_{p,i}^{\mathcal{A}} \cup (R_p \cap C_p^{\mathcal{A}})) \cap I_p$$

Accordingly, choose a state feedback map $\phi_p : C_p^{\mathcal{A}} \to \mathbb{C}$ so that for $x \in C_p^{\mathcal{A}}$ and $\sigma \in \phi_p(x)$, $\delta(\sigma, x) \in C_p^{\mathcal{A}}$, and for some $\sigma' \in \phi_p(x)$, $\delta(\sigma', x) \in C_{p,i}^{\mathcal{A}} \cup (R_p \cap C_p^{\mathcal{A}})$, where i is the least i such that $x \in C_{p,i+1}^{\mathcal{A}}$.

Now consider the entire subset $C^{\mathcal{A}} := \rho^{\mathcal{A}}(\bigcup_{p\in P} C_p^{\mathcal{A}}) = \bigcup_{i=1}^{n} C_i^{\mathcal{A}}$, where

$$C_0^{\mathcal{A}} := \emptyset; \text{ and, for } i > 0,$$
$$C_{i+1}^{\mathcal{A}} := \theta^{\mathcal{A}(\not\to C^{\mathcal{A}})}(C_i \cup \textstyle\bigcup_{p\in P} C_p^{\mathcal{A}})$$

Define, for each $C_{i+1}^{\mathcal{A}}$, a state feedback map $\phi_{i+1} : C_{i+1}^{\mathcal{A}} \longrightarrow \mathbb{C}$ such that for all $x \in C_{i+1}^{\mathcal{A}}$, $\delta(\sigma, x) \in C^{\mathcal{A}}$, for all $\sigma \in \phi_{i+1}(x)$; and for some $\sigma' \in \phi_{i+1}(x)$, $\delta(\sigma', x) \in C_i^{\mathcal{A}} \cup \bigcup_{p\in P} C_p^{\mathcal{A}}$.

Finally, choose an arbitrary total ordering of the index set P and define the state feedback map $\phi : C^{\mathcal{A}} \longrightarrow \mathbb{C}$ such that

i. if $x \in C^{\mathcal{A}} \setminus \bigcup_{p\in P} C_p^{\mathcal{A}}$, then $\phi(x) = \phi_i(x)$, for the least i such that $x \in C_i^{\mathcal{A}}$;

ii. and if $x \in \bigcup_{p\in P} C_p^{\mathcal{A}}$, then $\phi(x) = \phi_p(x)$, for the least $p \in P$ such that $x \in C_p^{\mathcal{A}}$.

Now define, for any $x \in C^{\mathcal{A}}$, the map

$$\begin{aligned} f_x :\ & \Sigma^* \longrightarrow \mathbb{C} \\ & k \longrightarrow \phi(\delta(k, x)) \end{aligned}$$

We must show that f_x satisfies both clauses of the definition of $F^{\mathcal{A}}$. The first clause follows directly from the definition of $\phi : C^{\mathcal{A}} \longrightarrow \mathbb{C}$. For the second clause, suppose that $s \in \Sigma^\omega$ is genereated by $\mathcal{A}_x$ under f_x. Note first that $\delta(k, x) \in C^{\mathcal{A}}$, for any prefix k of s. Now if s has infinitely many prefixes k such that $\delta(k, x) \in C_{i+1}^{\mathcal{A}} \setminus \bigcup_{p\in P} C_p^{\mathcal{A}}$, then, by the definition of ϕ_{i+1} and the definition of generated ω-strings, s must have infinitely many prefixes k' such that $\delta(k', x) \in C_i^{\mathcal{A}} \cup \bigcup_{p\in P} C_p^{\mathcal{A}}$. Since i is arbitrary, it follows that s has a prefix $k'' \in \Sigma^*$ such that $\delta(k'', x) \in \bigcup_{p\in P} C_p^{\mathcal{A}}$.

For any $x \in \bigcup_{p\in P} C_p^{\mathcal{A}}$, let the *P-rank* of x be the least index $p \in P$ such that $x \in C_p^{\mathcal{A}}$ and let the *rank* of x be the least pair (p, i) in the lexicographic ordering of $P \times \mathbb{N}$ such that $x \in C_{p,i}^{\mathcal{A}}$. It follows by the definition of ϕ that if $k'' \in C_p^{\mathcal{A}}$, then the P-rank of $\delta(k''', x)$ is no greater than that of $\delta(k'', x)$, for all $k'' \leq k''' < s$. Therefore, let q be the least $q \in P$ such that $\delta(l, x) \in C_q^{\mathcal{A}}$ for some (and therefore for co-finitely many) $l < s$. It follows from the definition of ϕ_q and the definition of the generation of ω-strings that if s has infinitely many prefixes l' such that $\delta(l', x)$ has rank (q, j) then s has infinitely many prefixes l'' such that either the rank of $\delta(l'', x)$ is less than (q, j) or $\delta(l'', x) \in R_q \cap C_q^{\mathcal{A}}$. Thus s must have infinitely many prefixes $k \in \Sigma^*$ such that $\delta(k, x) \in R_q$ and only finitely many such that $\delta(k, x) \notin I_q$. It follows that s is accepted by $\mathcal{A}_x$. This establishes the inclusion $F^{\mathcal{A}} \supseteq C^{\mathcal{A}}$.

For the reverse inclusion, $F^{\mathcal{A}} \subseteq C^{\mathcal{A}}$, we use induction on the number of *live* states of $\mathcal{A}$. A state is called live if there exists a transition leading from that state to another (different) state. The set of live states of $\mathcal{A}$ is denoted $L(\mathcal{A})$. An approximate opposite to

liveness is *degeneracy*. A state is degenerate if there do exist transitions that are defined for that state, but all such transitions lead only to the state itself. Thus the sets of live and degenerate states are disjoint, but their union need not equal the entire state set. The set of degenerate states of $\mathcal{A}$ is denoted $D(\mathcal{A})$. Note that the operations of self-looping and restriction potentially convert live states to degenerate states.

It is easy to see that $F^{\mathcal{A}}, C^{\mathcal{A}} \subseteq L(\mathcal{A}) \cup D(\mathcal{A})$ and that

$$F^{\mathcal{A}} \cap D(\mathcal{A}) = \bigcup_{p \in P} (R_p \cap I_p \cap D(\mathcal{A})) = C^{\mathcal{A}} \cap D(\mathcal{A})$$

Thus it suffices to show that $F^{\mathcal{A}} \cap L(\mathcal{A}) \subseteq C^{\mathcal{A}}$. If $L(\mathcal{A}) = \emptyset$ then this inclusion holds vacuously. Suppose therefore that $x \in F^{\mathcal{A}} \cap L(\mathcal{A})$ and make the inductive hypothesis that the result holds for all automata with fewer live states than $\mathcal{A}$. By definition of $F^{\mathcal{A}}$, there exists a feedback map f that controls $\mathcal{A}$ to satisfy its acceptance condition in deadlock-free fashion. We perform a case analysis on the closed-loop behaviour of $\mathcal{A}$ under the map f:

i. if there exists some $x' \in L(\mathcal{A})$ such that $\delta(k, x) \neq x'$ for all $k \in \Sigma^*$ generated by $\mathcal{A}_x$ under f, then

$$\begin{array}{rcll} x & \in & F^{\mathcal{A}\restriction(X \setminus \{x'\}))} & \\ & \subseteq & C^{\mathcal{A}\restriction(X \setminus \{x'\}))} & \text{(ind. hyp.)} \\ & \subseteq & C^{\mathcal{A}} & \text{(by Prop. 3.2)} \end{array}$$

ii. if there exist some $x', x'' \in L(\mathcal{A})$ and some $k \in \Sigma^*$ such that $\delta(k, x) = x'$ and $\delta(k', x) \neq x''$ for all $k' \geq k$ generated under f, then by the above argument, $x' \in C^{\mathcal{A}}$, and so,

$$\begin{array}{rcll} x & \in & F^{\mathcal{A}(\hookrightarrow \{x'\})} & \\ & \subseteq & C^{\mathcal{A}(\hookrightarrow \{x'\})} & \text{(ind. hyp.)} \\ & = & C^{\mathcal{A}} & \text{(Prop. 3.2)} \end{array}$$

iii. finally, if for all $x', x'' \in L(\mathcal{A})$ and $k \in \Sigma^*$ such that $\delta(k, x) = x'$, there exists $k' \geq k$ generated by $\mathcal{A}_x$ under f, such that $\delta(k', x) = x''$ then

(a) if for every $x' \in L(\mathcal{A})$ and $k \in \Sigma^*$ generated by $\mathcal{A}_x$ under f such that $\delta(k, x) = x'$, there exists $k' > k$ generated by $\mathcal{A}_x$ under f such that $\delta(k', x) \in C^{\mathcal{A}} \cap D(\mathcal{A})$, then

$$L(\mathcal{A}) \subseteq \mu X_1.\ \theta^{\mathcal{A}(\not\to L(\mathcal{A}))}(X_1 \cup (C^{\mathcal{A}} \cap D(\mathcal{A})))$$

(by Theorem A.1 and induction on the length of strings), so $L(\mathcal{A}) \subseteq \rho^{\mathcal{A}}(C^{\mathcal{A}}) = C^{\mathcal{A}}$;

(b) otherwise we can show that there exists a string $s \in \Sigma^\omega$ generated by $\mathcal{A}_x$ under f such that for any $x' \in L(\mathcal{A})$, there exists $k < s$ such that $\delta(k, x) = x'$. It follows that $L(\mathcal{A}) \subseteq I_p$ and $L(\mathcal{A}) \cap R_p \neq \emptyset$, and therefore

$$L(\mathcal{A}) \subseteq \mu X_1.\ [\theta^{\mathcal{A}(\not\to L(\mathcal{A}))}(X_1 \cup (R_p \cap L(\mathcal{A}))) \cap I_p]$$

(by Theorem A.1), for some $p \in P$. It follows that $L(\mathcal{A}) \subseteq C_p^{\mathcal{A}} \subseteq C^{\mathcal{A}}$ (by Theorem A.1).

□

4 Conclusion

We have considered a version of the control problems of [TW94a, Thi] that admits a polynomial-time solution.

The problem considered in [TW94a] is formally equivalent to Church's problem and the emptiness problem for automata on infinite trees, and that studied in [Thi] is equivalent to extended versions of the the above problems. The present problem is a counterpart of the probabilistic tree-automaton emptiness problem studied in [CY88]. Indeed, the solution presented here when applied to controlled probabilistic automata can be viewed as yielding a controlled behaviour satisfying the acceptance conditions with probability one.

References

[CVW86] Constantin Courcoubetis, Moshe Y. Vardi, and Pierre Wolper. Reasoning about fair concurrent programs (extended abstract). In *Symposium on the Theory of Computing*, pages 283–294. ACM, 1986.

[CY88] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 338–345, 1988.

[EJ88] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *29th Annual Symposium on Foundations of Computer Science*, pages 328 – 337, 1988.

[EL86] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Symposium on Logic in Computer Science*, pages 267 – 278. IEEE, June 1986.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 179–190. Association for Computing Machinery, January 1989.

[Ram89] Peter J. G. Ramadge. Some tractable supervisory control problems for discrete-event systems modeled by Büchi automata. *IEEE Trans. Automatic Control*, 34(1):10–19, January 1989.

[RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[Thi] J. G. Thistle. On control of systems modelled as deterministic Rabin automata. Submitted for publication.

[TW94a] J. G. Thistle and W. M. Wonham. Control of infinite behaviour of finite automata. To appear in *SIAM J. Control and Optimization* **32** (3), 1994.

[TW94b] J. G. Thistle and W. M. Wonham. Supervision of infinite behaviour of discrete-event systems. To appear in *SIAM J. Control and Optimization* **32** (3), 1994.

[WW93] K.C. Wong and W. M. Wonham. Hierarchical control of timed discrete-event systems. In *Proceedings of Second European Control Conference*, pages 509–512, June-July 1993.

A Fixpoint preliminaries

Our definitions of monotonicity and continuity and our presentation of the basic results of Tarski and Knaster (Theorem A.1) are adapted from those of [EL86].

A k-ary *operator* on a power set 2^X is a map $f : (2^X)^k \to 2^X$.

An operator is *monotone* if it preserves inclusion; that is,

$$X_i \subseteq X_i' \Longrightarrow f(X_1, \ldots, X_i, \ldots X_k) \subseteq f(X_1, \ldots, X_i', \ldots X_k) \ , \ 1 \leq i \leq k \ .$$

An operator is $\cup$-*continuous* if, for any i, $1 \leq i \leq k$, and any nondecreasing sequence $X_i^0 \subseteq X_i^1 \subseteq X_i^2 \ldots$,

$$\bigcup_{j=0}^{\infty} f(X_1, \ldots, X_i^j, \ldots, X_k) = f(X_1, \ldots, \bigcup_{j=0}^{\infty} X_i^j, \ldots, X_k)$$

An operator is $\cap$-*continuous* if, for any i, $1 \leq i \leq k$ and any nonincreasing sequence $X_i^0 \supseteq X_i^1 \supseteq X_i^2 \ldots$,

$$\bigcap_{j=0}^{\infty} f(X_1, \ldots, X_i^j, \ldots, X_k) = f(X_1, \ldots, \bigcap_{j=0}^{\infty} X_i^j, \ldots, X_k)$$

Both $\cup$-continuity and $\cap$-continuity imply monotonicity; for operators on finite power sets the reverse implications hold.

We denote extremal fixpoints of monotone operators by means of the "fixpoint quantifiers" μ and ν, which quantify over subsets. Expressions of the form

$$\mu Y.\ \phi(Y) \quad (\text{resp. } \nu Y.\ \phi(Y))$$

represent the least (resp. greatest) $Y \subseteq X$ such that $Y = \phi(Y)$ – in other words the least (resp. greatest) fixpoints of the operator that maps every $Y \subseteq X$ to $\phi(Y)$. The question of the existence of such fixpoints is dealt with below.

Theorem A.1 (Tarski-Knaster)

Let $f : 2^X \to 2^X$ be a monotone operator on X. Then f has least and greatest fixpoints; in fact,

(i) $\mu Y.\ f(Y) \ = \ \bigcap\{Y' \subseteq X : \ Y' = f(Y')\} \ = \ \bigcap\{Y' \subseteq X : \ Y' \supseteq f(Y')\}$

(i') $\nu Y.\ f(Y) \ = \ \bigcup\{Y' \subseteq X : \ Y' = f(Y')\} \ = \ \bigcup\{Y' \subseteq X : \ Y' \subseteq f(Y')\}$

(ii) if f is $\cup$-continuous then $\mu Y.\ f(Y) = \bigcup_{i=0}^{\infty} f^i(\emptyset)$.

(ii') if f is $\cap$-continuous then $\nu Y.\ f(Y) = \bigcap_{i=0}^{\infty} f^i(X)$.

(where f^i denotes the i-fold composition of f with itself).

□

An Introduction to a Synchronized Petri Net Based Tool for the Synthesis of Supervisors of Discrete Event Systems[1]

J.-M. Palmier, M. Makungu, F. E. Agapi, M. Barbeau, and R. St-Denis

Département de mathématiques et d'informatique
Université de Sherbrooke
Sherbrooke (Québec)
CANADA J1K 2R1

Abstract

This work is part of a project which consists of developing methodologies for automatic synthesis of supervisors of discrete event systems. In this paper, we present a software tool which allows synthesis of supervisors of industrial processes modeled as Synchronized Petri Nets (SPNs). Control requirements are also expressed as SPNs. This software tool is based on the theoretical framework of Ramadge and Wonham. The paper describes a connection between SPNs and the framework of Ramadge and Wonham. Our software tool uses Design/CPN for graphic edition of SPNs. Synthesis algorithms are implemented in C++. Implementation issues are further discussed in the paper.

1. Introduction

The modeling power of automata is sometimes limited, and more and more engineers rather use Petri nets for modeling their processes. Indeed, Petri nets generally lead to smaller models than equivalent automaton representations. We are developing a software tool which allows generation of supervisors of plants modeled as Synchronized Petri Nets (SPNs) [Moal 85]. The tool is based on the theoretical framework for synthesis of supervisors of Ramadge and Wonham [Rama 89]. The problem addressed by this framework is construction a supervisor C which task consists of enabling and disabling events of given system components in accordance with certain requirements. The framework of Ramadge and Wonham requires at the specification level: i) the development of a model G of the plant; ii) the identification of a subset Σ_u of the events Σ in G that are uncontrollable; iii) the description of a mask function $\Pi : \Sigma \rightarrow \Lambda \cup \{ \varepsilon \}$ which represents how the events in G are observed by a supervisor (where Λ is the set of observed events); and iv) the expression of the control requirements as a legal language L (the behavior of C coupled with G must be contained in L).

[1]The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds pour la formation de chercheurs et l'aide à la recherche (FCAR).

In our approach, the plant model G and the model of the legal language L are both expressed as SPNs. The connection between SPNs and the framework of Ramadge and Wonham consists of the following. First, reachability graphs of the SPNs are computed. The resulting graphs are automaton representations of the language of the plant and the legal language. Second, these automata become the input of a set of C++ classes that implements the supervisor synthesis framework of Ramadge and Wonham. The construction of real size SPNs cannot be realized without using a tool that provides an easy and efficient way to edit and manipulate Petri nets. The development of such a tool represents a big amount of effort and, instead of programming our own tool, we interfaced our environment with a commercial product called Design/CPN [Meta 93].

This paper is structured as follows. Section 2 discusses the theoretical foundations of the approach. An overview of the functionalities of the tool is presented in Section 3 and its implementation is discussed in Section 4. Finally, concluding remarks are presented in Section 5.

2. Theoretical Foundations

Hereafter, we adopt the terminology and notation conventions from [Bram 83] and [Moal 85]. The basic problem in supervisory control is to construct a supervisor whose task is to enable and disable the controllable events of a discrete event system.

Discrete Event System

A Discrete Event System (DES), also called a plant, is modeled as a bounded Synchronized Petri Net (SPN):

$$G = \langle P, T, Pre, Post, \Sigma, \mu, Cap, M_0 \rangle$$

that is: i) a Petri net $\langle P, T, Pre, Post \rangle$, where P is a bounded set of places, T a bounded set of transitions, $Pre: P \times T \rightarrow N$ (N denotes the set of natural numbers) a backward incidence function, and $Post: P \times T \rightarrow N$ a forward incidence function; ii) a set Σ of asynchronous and discrete events; iii) a transition labeling function $\mu: T \rightarrow \Sigma$ that associates to each transition in T an event in Σ; iv) a function $Cap: P \rightarrow N - \{0\}$ that associates a capacity to every place; and finally, iv) an initial marking $M_0: P \rightarrow N$.

A DES is pictured in Fig. 1 a). Places are shown as circles, that is, $P=\{p_1,p_2,p_3,p_4\}$, and transitions as rectangles, that is, $T=\{t_1,t_2,t_3,t_4\}$. The backward incidence function is pictured as arrows from places to transitions, e.g., $Pre(p_1,t_1)=1$ and

$Pre(p_1,t_3)=0$. The forward incidence function is pictured as arrows from transitions to places, e.g., $Post(p_2,t_1)=1$. The labeling of a transition, that is, $\mu(\cdot)$, is inscribed in the transition rectangle. All places have capacity 2, that is, $Cap(p)=2$ for all $p \in P$. The initial marking, shown as integers inside places, is $M(p1)=M(p3)=1$ and $M(p2)=M(p4)=0$. This simple example clearly illustrates the compactness of a Petri net model with respect to an equivalent automaton model. Indeed, a single Petri net structure is shared by two concurrent components. The same behavior expressed as automata would require two state-transition machines.

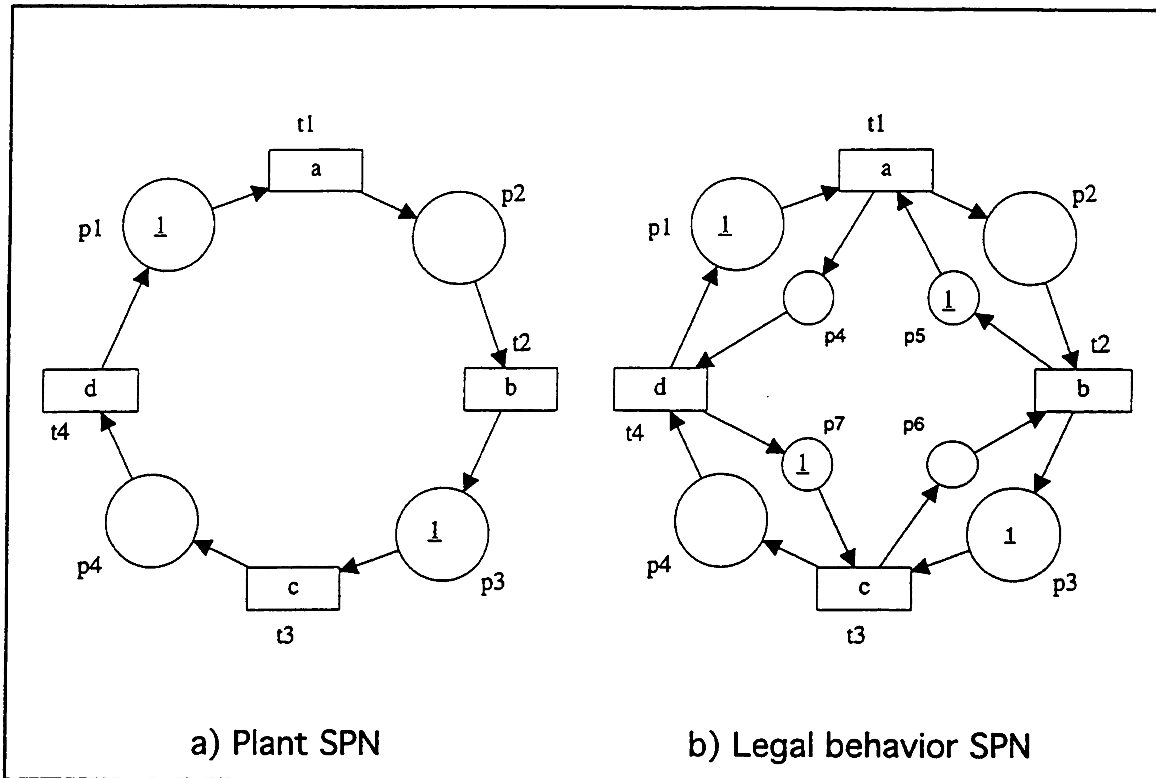


Figure 1: Plant SPN and legal behavior SPN

A transition $t \in T$ is enabled in a marking M if $M(p) \geq Pre(p,t)$ and

$$M(p)-Pre(p,t)+Post(p,t) \leq Cap(p) \text{ for all } p \in P.$$

An enabled transition may occur resulting in a new marking M' defined as:

$$M'(p)=M(p) - Pre(p,t) + Post(p,t) \text{ for all } p \in P.$$

The fact that marking M' is reachable from marking M after the occurrence of transition t is denoted as $M[t>M'$. This notation is naturally extended to sequences of transitions $\tau \in T^*$,

that is, $M[\tau>M'$ means that marking M' is reachable from marking M after the occurrence of the sequence of transitions τ.

We denote as L(G) the language generated by G which is defined as:

$$L(G) = \{ s\in \Sigma^* \mid \text{there exist } \tau\in T^* \text{ and } M:P\rightarrow N \text{ such that } \mu(\tau)=s \text{ and } M_0[\tau>M \}.$$

For the DES of Fig. 1 a),

$$L(G) = \overline{(abcd)^*} \,\|\, \overline{(cdab)^*}.$$

where $\|$ denotes the shuffle product operator [Gins 66] and the overline bar represents the prefix closure operator.

The set of all strings of events is denoted as Σ^*. The empty string is represented as ε. A subset $L\subseteq\Sigma^*$ is called a language over Σ. The prefix closure of L is denoted as $\overline{L}$ and is the set of all strings that are prefixes of strings in L. If $L=\overline{L}$, then L is said to be *prefix closed.*

Controlled Discrete Event System

The set Σ of a DES is partitioned into $\Sigma=\Sigma_u\cup\Sigma_c$, the sets of uncontrollable and controllable events. For the plant of Fig. 1 a), the set of uncontrollable events is $\Sigma_u=\{b\}$ whereas the set of controllable events is $\Sigma_c=\{a,c,d\}$. Let $\Gamma=\{0, 1\}^\Sigma$ be the set of all binary assignments to the elements of Σ. A controlled DES (CDES) is a SPN:

$$G_c = < P, T, Pre, Post, \Sigma, \mu, Cap, M_0, \Gamma >$$

where a transition t may occur in a marking M with an event $\sigma\in\Sigma$ and a control vector $\gamma\in\Gamma$ if and only if: i) t is enabled in marking M, ii) $\mu(t)=\sigma$, and iii) the control vector enables the event labeling transition t, that is, $\gamma(\sigma)=1$.

Supervised Discrete Event System

A supervisor $C=(S,\phi)$ is a pair which includes a deterministic automaton S and a feedback function ϕ defined as:

$$S = (X, \Lambda, \xi, x_0, X_m) \text{ and } \phi : X \rightarrow \Gamma$$

S must be interpreted as a device driven by the sequence of events generated by G, while the behavior of G must be controlled by the function ϕ, and thus indirectly by the states of S. If $\phi(x)(\sigma)=\gamma(\sigma)=1$, then σ is enabled; otherwise, σ is disabled.

In some cases, C cannot observe all the events of G_c. In others, it cannot distinguish between certain events. These cases are resolved by introducing an observation component, between G_c and C, represented by a mask function:

$$\Pi : \Sigma \rightarrow \Lambda \cup \{ \varepsilon \}$$

where $\lambda=\Pi(\sigma)$ is the event of G_c observed by C when Π masks the event σ. Thus, the events in $\Pi^{-1}(\varepsilon)$ are those that cannot be seen by C. If C cannot distinguish between σ_1 and σ_2, then $\Pi(\sigma_1)=\Pi(\sigma_2)$. If Π is the identity function, then $\Lambda=\Sigma$ and the original events are all observed by C. In Fig. 1 a), all events are indentically observed but "a" is observed as ε, that is, $\Pi(a)=\varepsilon$, $\Pi(b)=b$, $\Pi(c)=c$, and $\Pi(d)=d$.

A supervised DES (SDES) is a pair consisting of a CDES G_c and a supervisor C represented together as:

$$C/G_c = (G_c, C) = (< P, T, Pre, Post, \Sigma, \mu, Cap, M_0, \Gamma > , (S, \phi))$$

where C/G_c means that G_c is controlled by C. The transition function of C/G_c is defined as follows. Let x and x' denote states of S and M and M' denote markings of G_c. A transition of C/G_c from a state (x,M) on event $\sigma \in \Sigma$ to a state (x',M') is represented as:

$$(x,M) \xrightarrow{\sigma} (x',M')$$

and defined if and only if: i) there exists a transition $t \in T$ such that $\mu(t)=\sigma$ and M[t>M', ii) $\xi(\Pi(\sigma), x)$ is defined and $\xi(\Pi(\sigma), x) = x'$, and iii) $\phi(x)(\sigma)=1$.

The control requirements are captured by the notion of legal behavior L. The legal behavior L is a language, modeled as a SPN, which elements are the admitted event sequences. For the example of Fig. 1, the legal behavior is shown in part b). In this example, the plant may generate non admitted event sequences, e.g., the word "ab" is in L(G) and not in L.

A supervisor synthesis problem is stated as follows. Given a CDES G_c and its legal behavior L, find a supervisor $C=(S,\phi)$ such that $L(C/G_c) \subseteq L$ is as large as possible. In this paper, the plant behavior as well as the legal behavior are modeled as SPNs. A

supervisor is represented as an automaton. Synchronous operation of the plant and supervisor behaviors, as shown above, restricts the plant to legal sequences of events. The reader may be interested to look at related models such as the one of Kumar and Holloway were both the plant and the supervisor are modeled as Petri nets [Kuma 92].

3. Overview of the Tool

The generation of a supervisor for a plant modeled by a SPN, with our software tool, consists of four steps: i) edition of the SPNs of the plant behavior and the legal behavior, ii) computation of reachability graphs of both SPNs, and iii) generation of a supervisor. These steps are discussed hereafter in more details.

Edition of the SPN

The SPNs modeling the plant and the legal language are constructed using Design/CPN, a package developed by Meta Software Corporation [Meta 93]. This software supports edition and simulation of Hierarchical Colored Petri Nets [Jens 93]. It provides everything required to edit the SPNs of our application, so we did not have to program our own editing tool. The SPNs of Fig. 1 have been created using Design/CPN.

Unfortunately, Design/CPN does not provide direct access to the internal representation of edited Petri nets. Therefore, we programmed in ML a function that creates a file containing the description of a SPN. ML is the programming language of Design/CPN and ML programs can be executed within the Design/CPN environment.

Computation of reachability graphs

Computation of reachability graphs of the plant SPN as well as the legal language SPN is required for the generation of the supervisor. The reachability graph of a SPN is an automaton which states are the reachable markings of the SPN and transitions are occurrences of transitions of the SPN [Bram 83].

Computation of reachability graphs is done by a program written in C++. It uses data structures compatible with those used in the next phase of the synthesis process. Our program is inspired by a Pascal program called Trecon developed at Aarhus University [Jens 87].

Generation of the supervisor

The supervisor synthesis method [Barb 93] borrows algorithms from Ramadge and Wonham [Rama 87, Wonh 87] and Cieslak et al. [Cies 88]. It also exploits a new algorithm for handling partially observed plants [Barb 94].

The inputs of the method are: i) an automaton generating the plant language, ii) an automaton generating the legal language L, iii) a set Σ_u of uncontrollable events, and iv) a mask function $\Pi:\Sigma\rightarrow\Lambda\cup\{\varepsilon\}$. The output is a supervisor $C=(S,\phi)$ where S is an automaton and $\phi:X\rightarrow\Gamma$ a feedback function. The supervisor is synthesized such that two properties are satisfied by the SDES C/G_c. First, C/G_c must generate solely legal sequences of events, that is $L(C/G_c)\subseteq L$. Second, since some events may be indistinguishable or unobservable, it is needed that the supervisor behaves in the same way on sequences of events that cannot be distinguished, for obvious consistency reasons. These required properties are satisfied by synthesizing a supervisor C such that the language of C/G_c is a sublanguage K_R of L with three formal properties. Namely, K_R is:

- prefix closed, that is, $K_R = \overline{K}_R$;
- $(\Sigma_u, L(G))$-invariant, that is, $s\in \overline{K}_R$, $\sigma\in\Sigma_u$, $s\sigma\in L(G)\Rightarrow s\sigma\in \overline{K}_R$; and
- $(\Pi, L(G))$-recognizable, that is, $s\in K_R$, $s'\in L(G)$, $\Pi(s)=\Pi(s')\Rightarrow s'\in K_R$.

4. Implementation of the Synthesis Algorithms

This section described implementation of the algorithms for the synthesis of supervisors. Supervisors can be derived using different algorithms [Wonh 87, Cies 88, Cho 89, Wonh 93, Barb 94]. The choice of an algorithm depends on the considered problem. In order to carry out synthesis of supervisors, operational forms of algorithms are required. A natural approach to achieve greater efficiency in the implementation of such algorithms is that of object-oriented programming, since the mathematical objets can be considered as abstract data types. Object-oriented programming have three obvious advantages. The first is that a class is simply a model from which each object is created with its own attributes that no other object can access except through the local services of the former. The second advantage is that, through the mechanism of inheritance, subclasses automatically share all attributes and services of their superclasses. Finally, a major improvement over procedural programming is that a uniform interface can be provided over a wide range of object types. The same service name can be used for different objects, differently implemented. However, these differences

are transparent to the clients. Therefore, we implemented objects and algorithms handled in the synthesis process as a kernel of basic C++ classes that can be incrementally modified and extended. The kernel is powerful and simple enough to be used with several logical models of discrete event systems. The model currently implemented is automata and direct support for Petri nets is planned.

Figure 2: Class diagram

Fig. 2 illustrates the structure of the main classes included in the kernel. The notation used is an extension of Coad and Yourdon's notation [Coad 91]. The primitive classes are *AState* (atomic state), *CState* (composite state), *Event*, and *Transition* . The first two are subclasses of class *State*. A composite state is built from a tuple of atomic states. Every object of class *State* has both an internal compact representation and an external symbolic representation. The former is required for efficient implementation of algorithms. The latter allows the user to provide input which is closer to a natural language and to read results in an uncryptic manner, thus facilitating traceability between inputs and outputs. Furthermore, symbolic reasoning tools that generate legal languages [Larri 94] or establish diagnostics on the way solutions were obtained can profits of such a framework. To illustrate this point, let us consider the case where the solution obtained is the empty supervisor. This case arises when the safety properties cannot be satisfied without pruning all states of the automaton representing the legal language during the synthesis process [Wonh 87], or when the liveness properties cannot be fulfilled during the analysis of the control-loop system with model checking algorithms [Clar 92]. In such case, the initial constraints must be weakened or the behavior of some plant components need to be changed. Minimal and optimal modifications to the original problem cannot, however, be done without a powerful symbolic diagnostic tool.

Fig. 2 also shows a generic class, called *Mapping*, provided to define various king of mapping such as feedback, mask, and correspondence functions. Finally, the class *Automaton* its instances are defined as aggregations of objects of primitive classes. Each automaton object contains an initial state, a set of states, a set of events, and a transition graph.

Fig. 3 shows the interface definition of the class *Supervisor*. It provides synthesis services as constructors creating objects of the class *Supervisor*. Based on a variety of polymorphism, called overloading, synthesis of a supervisor is completely transparent to the user, since each constructor has its own parameter list. For example, the first constructor is used when events are identically observed and the correspondence function between the states of the plant and those of the legal language is unknown [Wonh 87]. This indicates that the correspondence function must be calculated before the derivation of the supervisor. In some problems, the correspondence function is known (it can be the identity function). In this case, the second constructor is used. Finally, the next two constructors are used in a similar fashion, but to solve the supervisory control problem under partial observation [Cies 88, Cho 89, Barb 94].

```
class Supervisor
{
  private:
          Automaton s;
          Feedback phi;

  public:
          Supervisor(Automaton plant, Automaton legal);
          Supervisor(Automaton plant, Automaton legal, Correspondence f);
          Supervisor(Automaton plant, Automaton legal, Mask m);
          Supervisor(Automaton plant, Automaton legal, Mask m, Correspondence f);
.
.
.
}
```

Figure 3: Definition of the class *Supervisor*

5. Conclusion

This paper has introduced a Synchronized Petri Net (SPN) based software tool for the systematic construction of supervisor. Petri nets are interesting because they lead to more compact models than automaton models. Petri net models are therefore easier to understand and less complex to analyze. Reachability graphs of SPNs are computed and the synthesis process is carried out. A C++ implementation of the synthesis algorithms has been briefly described. This work is being done to support a methodology for the specification and development of process-control systems [Barb 93].

References

[Barb 93] M. Barbeau and R. St-Denis, "A Rigorous approach for the specification and development of process-control systems", *Proceedings of the 6th International Conference of Software Engineering & its Applications*, Paris, November 1993, 599-608.

[Barb 94] M. Barbeau, G. Custeau, and R. St-Denis, "An Algorithm for computing the mask value of the supremal normal sublanguage of a legal language", Technical report no. 125, Département de mathématiques et d'informatique, Université de Sherbrooke, January 1994.

[Bram 83] G. W. Brams, *Réseau de Petri : théorie et pratique*, éditions MASSON, 1983.

[Cho 89] H. Cho and S. I. Marcus, "On supremal languages of classes of sublanguages that arise in supervisor synthesis problems with partial observation", *Mathematics of Control, Signals, and Systems*, 2, 1989, 47-69.

[Cies 88] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya, "Supervisory control of discrete-event processes with partial observations", *IEEE Transactions on Automatic Control,* 33 (3), 1988, 249-260.

[Clar 92] E. M. Clarke, O. Grumberg, and D. E Long, "Model checking and abstraction", *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, January 1992, 343-354.

[Coad 91] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Second Edition, Yourdon Press Computing Series, 1991.

[Gins 66] S. Ginsburg, *The Mathematical Theory of Context Free Languages*, McGraw-Hill Book Company, 1966.

[Jens 87] K. Jensen, P. Huber, P. O. Jensen, N. N. Larsen, and I.-M. Martinsen, "Petri net package program documentation", Version 4.1, Technical Report from Computer Science Department, Aarhus University, May 1987.

[Jens 93] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[Kuma 92] R. Kumar and L. E. Holloway, "Supervisory control of Petri net languages", *Proceedings of the 31st Conference on Decision and Control,* Tucson, Arizona, December 1992, 1190-1195.

[Larri 94] P. Larivière, "Dérivation de langages légaux à partir de contraintes exprimées en logique temporelle", Mémoire de maîtrise, Département de mathématiques et d'informatique, Université de Sherbrooke, à paraître.

[Meta 93] Meta Software Corporation, *Design/CPN Reference Manual for X-Windows - Version 2.0*, 125 Cambridge Park Drive, Cambridge, MA 02140 USA, 1993.

[Moal 85] M. Moalla, "Réseaux de Petri interprétés et Grafcet", *Technique et Science Informatiques*, No. 1, 1985, 17-30.

[Rama 87] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes", *SIAM Journal Control and Optimization*, 25 (1) 1987, 206-230.

[Rama 89] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems", *Proceedings of the IEEE*, Vol. 77, No. 1, January 1989, 81-98.

[Wonh 87] W. M. Wonham and P. J. Ramadge, "On the supremal controllable sublanguage of a given language", *SIAM Journal Control and Optimization*, 25 (3) 1987, 637-659.

[Wonh 93] W. M. Wonham, "Notes on control of discrete-event systems", Department of Electrical Engineering, University of Toronto, June 1993.

Hierarchical COCOLOG of a Finite Machine

Y.J. Wei Peter E. Caines*

Department of Electrical Engineering, McGill University
3480 University Street, Montreal, P.Q, Canada H3A 2A7
*The Canadian Institute for Advanced Research, Canada
peterc@cim.mcgill.edu

Abstract

The theory of the dynamically consistent partition lattice of a given finite machine $\mathcal{M}$ is briefly reviewed. A family of multiple sort first-order logics called hierarchical COCOLOG is introduced to logically express mutil-level control theories for $\mathcal{M}$. During a (two-level) realization of a control strategy for $\mathcal{M}$ solving a given control task, communication takes place between the COCOLOG theories at each level; this communication is mediated through two extra-logical rules called respectively *the instantiation rule* and *the block membership rule.*

Keywords: discrete event systems, finite machine, logical control, hierarchical structure, lattice.

1 Introduction

The apparent existence of hierarchical systems in the real world and the perception that hierarchical structures possess certain properties of efficient information processing and control have motivated many attempts to give mathematical formulation to the notion of a hierarchically structured system. We follow in the line of work in the mathematical theory of automata and input-state-output machines. Notable among many contributions in this content is Krohn-Rhodes decomposition theory; they proved that any finite automaton can be decomposed via covering by a cascade product of a list of permutation-reset automata [G68]; their work may be viewed as the generalization of Jordan-Holder theorem of group theory to the semi-group setting of automata theory. While the Krohn-Rhodes theory is expressed in terms of the covering of an automaton, the hierarchical lattice of a finite machine $\mathcal{M}$, which is presented in [CWei94] and used in this paper, is lattice structured decomposition of $\mathcal{M}$. A set of ideas relating to the present paper is to be found in the book by J.Hartmanis and R. Stearns [HS66].

Within the automata formulation of control theory for discrete event systems (DES), the research by Zhong and Wonham [ZW90] is devoted to the hierarchical structure of DES; one of the primary motivations for this work is the creation of aggregation methods which ensure the consistency of the dynamical controllable specification of events at the different levels of aggregation of a given automaton. The resulting theory has some points of contact with the present work in that the editing procedure generates the set of so-called dynamically consistent automata. Other work in the decomposition and decentralized control is to be found in Lin and Wonham [LW90], Rudie and Wonham [RW92]; one principle distinction between that work and the notion presented here is that the state of the overall controlled system in [LW90,RW92] is the product of the states of various sub-automata, where in this paper, the state of the system is the singleton $x \in X$ which falls in different submachines according to the state partition π and coordination.

2 The Hierarchical Lattice of a Finite Machine

Subsection 1 and 2 of this section review the formulation of the hierarchical lattice of a finite machine. Readers are reference of [CWei94] for a detailed presentation.

2.1 Dynamically Consistent Partitions and Partition Systems

In this paper, a finite system is modeled by a finite machine $\mathcal{M} =< X, U, \Phi >$, where X is the finite set of states, U is the finite set of inputs, $\Phi : X \times U \rightarrow X$ is the transition function. Our formulation of the hierarchical structure of a finite system is based upon nested partitions of the system's state space, given by the lattice of partitions of X.

A partition π of X is a collection of pairwise disjoint subsets (called blocks) of X such that the union of this collection equals to X. A partition π_1 is said to be *stronger than* a partition π_2 if every π_1−block is a subset of some π_2-block. It is wildly known that the greatest lower bound and the least upper bound of two partitions are defined by *intersection* and *chain union* respectively. Further the set $PAR(X)$ of all partitions of X forms a lattice with respect to set inclusion partial order and the intersection and chain union, which is bounded above and below by trivial partition π_{tr} and the identity partition π_{id} respectively. We shall use the notion $\overline{X}_i$ to emphasize that the set X_i is taken as a singleton.

For a given state $x \in X$, one apply a sequence σ of inputs called *control sequence*, the system generates a sequence t of states called *trajectory*. $Con(\mathcal{M})$ and $Tra(\mathcal{M})$ shall denote the set of all legal control sequences and trajectories respectively. Given $\pi = \{X_1, \cdots, X_p\} \in PAR(X)$, A sequence of states $t = x_1 \cdots x_{l-1}x_l \in Tra(\mathcal{M})$ is called an *internal trajectory with respect to* X_i if $t \subset X_i$; it is called a *direct trajectory from* X_i *to* X_j if $x_1 \cdots x_{l-1} \subset X_i$ and $x_l \in X_j$ for some $i \neq j$. $Tra(\pi)^i_i$ and $Tra(\pi)^j_i$ shall denote respectively the set of all internal (respectively direct) trajectories of $\mathcal{M}$ with respect to X_i and X_j. Any trajectory can be decomposed into a sequence of sub-trajectories, say $t_1t_2 \cdots t_m$ such that t_i's are either internal or direct trajectories.

For the mapping $\theta : X \times Con(\mathcal{M}) \to Tr(\mathcal{M})$,

$$\theta((x,\sigma)) = x_0x_1x_2 \cdots x_p \in Tra(\mathcal{M}),$$

where $\sigma = u_0u_1u_2 \cdots u_{p-1} \in Con(\mathcal{M})$ and $x_0 = x$, $x_i = \Phi(x_{i-1}, u_{i-1})$, $1 \leq i \leq p$, we can define the following sets: $U_i^i = \theta^{-1}(Tra(\pi)_i^i)$ and $U_i^j = \theta^{-1}(Tra(\pi)_i^j)$. These sets shall play the role of high level control inputs in the hierarchical dynamical partition system. Define $lengh((x,\sigma) = |\sigma|$, then obviously, the lengths of element of U_i^j maybe different.

Now we introduce a concept called dynamical consistency. By that we mean that a high level transition from a block X_i to X_j is possible only when all states in X_i can be driven to some state in X_j without passing through a third block. A partition such that this consistency obtains, is called dynamically consistent partition.

Definition 2.1 Dynamically Consistent Condition and Partition System (DCC). Given $\pi = \{\overline{X}_1, \cdots, \overline{X}_k\} \in PAR(X)$ and $(\overline{X}_i, \overline{X}_j) \in \pi \times \pi$, a *dynamically consistent condition* (DCC) holds for the set pair (X_i, X_j) if for each element x_i of X_i, there exists at least one direct trajectory from X_i to X_j and which has initial state x_i, i.e.

$$\forall x \in X_i, \exists \sigma \in U^*, \forall \sigma' < \sigma \; (\Phi(x,\sigma') \in X_i \wedge \Phi(x,\sigma) \in X_j), \quad i \neq j, \tag{1}$$

$$\forall x \in X_i, \exists \sigma \in U^*, \forall \sigma' \leq \sigma \; (\Phi(x,\sigma') \in X_i), \quad i = j, \tag{2}$$

where $\sigma' \leq \sigma$ means that σ' is the initial segment of σ. For $\mathcal{U} = \{\overline{U}_i^j;\ U_i^j = \theta^{-1}(Tra(\pi)_i^j), 1 \leq i,j \leq k\}$, the transition function $\Phi^\pi : \pi \times \mathcal{U} \to \pi$ of the *dynamically consistent partition machine* (DCPM) $\mathcal{M}^\pi = <\pi, \mathcal{U}, \Phi^\pi>$ is defined by the equality relation $\Phi^\pi(\overline{X}_i, \overline{U}_i^j) = \overline{X}_j$ when DCC holds for $(X_i, X_j), 1 \leq i,j \leq |\pi|$ and not defined when DCC fails for (X_i, X_j). $PAR(\mathcal{M})$ shall denote the set of all dynamically consistent partition machines of $\mathcal{M}$.

From now on, we always assume that any partition machine in consideration is a DCPM.

Definition 2.2 Ordering of DCPM of $\mathcal{M}$.
Given $\mathcal{M}^{\pi_1} = <\pi_1, \mathcal{U}, \Phi^{\pi_1}>$, $\mathcal{M}^{\pi_1} = <\pi_1, \mathcal{V}, \Phi^{\pi_2}> \in PAR(\mathcal{M})$. We say $\mathcal{M}^{\pi_2}$ is *weaker* than $\mathcal{M}^{\pi_1}$, written as $\mathcal{M}^{\pi_1} \preceq \mathcal{M}^{\pi_2}$, if $\pi_1 \leq \pi_2$.

The trajectories and control sequence can be concatenated in a consistent way. Let $s = x_1 \cdots x_p, t = y_1 \cdots y_q \in Tra(\mathcal{M})$, then *dynamically consistent concatenation* of s and t is the string st if there exists $u \in U$ such that $\Phi(x_p, u) = y_1$, s otherwise. For $A, B \subset Tra(\mathcal{M})$, the dynamically consistent concatenation of A and B is: $A \circ B \underline{\Delta} \{s \circ t; s \in A \& t \in B\} \subset AB$. For $w_1 = (x_0, \sigma_1), w_2 = (y_0, \sigma_2) \in X \times Con(\mathcal{M})$, if $\Phi(x_0, \sigma_1) = y_0$, then the *dynamically consistent concatenation of w_1 and w_2*, written as $w_1 \circ w_2$ is the element $(x_0, \sigma_1\sigma_2)$, (x, σ_i) otherwise. For $C, D \subset X \times Con(\mathcal{M})$, $C \circ D = \{w \circ v; w \in C, v \in D\}$.

It is easy to prove Φ^π defined above satisfies semi-group property under dynamically consistent concatenation. one may show that if $\pi_1 \preceq \pi_2$, then a π_2-control event is a collection of dynamical consistent concatenations of some π_1- control events.

Definition 2.3 For $\mathcal{M}^{\pi_1}$ and $\mathcal{M}^{\pi_2} \in PAR(\mathcal{M})$, define

$$\mathcal{M}^{\pi_1} \cap \mathcal{M}^{\pi_2} = \mathcal{M}^{\pi_1 \cap \pi_2}, \quad \mathcal{M}^{\pi_1} \cup^c \mathcal{M}^{\pi_2} = \mathcal{M}^{\pi_1 \cup^c \pi_2},$$

Immediately, we have $HIPAL(\mathcal{M}) \underline{\Delta} < PAR(\mathcal{M}), \cap, \cup^c, \preceq>$ forms a lattice, which called *dynamically consistent hierarchical lattice of $\mathcal{M}$*.

2.2 In-block Controllability Partition Systems and its Associated Lattice

For an analysis of hierarchical system behavior, one should not only consider the global dynamics between the state space of subsystem but also the local dynamics in each subsystem given by the partition element $X_i \in \pi$. This inspires us to consider the so-called in-block controllability, by which we mean subsystem controllability.

Definition 2.4 Controllable Finite Systems.
A system $\mathcal{M}$ is called *controllable* if for any $(x, y) \in X \times X$, there exists a control sequence $\sigma \in U^*$ which gives rise to a trajectory from x to y, i.e. $\forall x \in X, \forall y \in X, \exists s \in U^*, \ (\Phi(x, s) = y)$.

For partition system we have

Definition 2.5 In-block and Between-block Controllability.
$\mathcal{M}^\pi$ is called *in-block controllable* if every associated submachine $\mathcal{M}_i = < X_i, U_i, \Phi_{|X_i \times U_i} >$ of the base system is controllable, i.e.

$$\forall X_i \in \pi, \forall x, y \in X_i, \exists s \in U_i^*, \forall s' \leq s, \ (\Phi(x, s') \in X_i \wedge \Phi(x, s) = y).$$

$\mathcal{M}^\pi$ is called *between-block controllable* if

$$\forall \overline{X}_i, \overline{X}_i \in \pi, \exists S \in \mathcal{U}^* \ (\Phi^\pi(\overline{X}_i, S) = \overline{X}_j).$$

In other word, $\mathcal{M}^\pi$ is controllable as a finite system. $IBCP(\mathcal{M})$ (respectively, $BBCP(\mathcal{M})$) shall denote the set of all in-block controllable (respectively, between-block controllable) partition systems.

In this paper, we focus on the algebra structure of $IBCP(\mathcal{M})$. We have

Theorem 2.1 $IBCP(\mathcal{M})$ is closed under chain union.

From the above theorem, $\mathcal{M}^{\pi_1 \cup^c \pi_2}$ is the least upper bound of $\mathcal{M}^{\pi_1}$ and $\mathcal{M}^{\pi_2}$ in $IBCP(\mathcal{M})$. Unfortunately, intersection does not preserve in-block controllability. Hence $IBCP(\mathcal{M})$ cannot be a lattice with respect $\cap$ and $\cup^c$.

Definition 2.6 Greatest Lower Bound *(glb in $IBCP(\mathcal{M})$)*
Given a finite machine $\mathcal{M}$ and two in-block controllable DC partitions π_1 and π_2, define

$$\pi_1 \sqcap \pi_2 \triangleq \bigcup^{c} \{\pi'; \ \ \pi' \preceq \pi_1, \pi' \preceq \pi_2\}$$

and define $\mathcal{M}^{\pi_1} \sqcap \mathcal{M}^{\pi_2} = \mathcal{M}^{\pi_1 \sqcap \pi_2}$.

Theorem 2.2 If $\mathcal{M}$ controllable, then $< IBCP(\mathcal{M}), \sqcap, \cup^c, \preceq >$ forms a lattice.

An in-block controllable partition system $\mathcal{M}^\pi$ inherits the controllability from the base system.

Theorem 2.3 Let $\mathcal{M}^\pi \epsilon IBCP(\mathcal{M})$, then $\mathcal{M}^\pi$ is a controllable (i.e. between-block controllable) if and only $\mathcal{M}$ is controllable.

Definition 2.7 The tuple $< \mathcal{M}^\pi, \mathcal{M}_1, \cdots, \mathcal{M}_p, \psi >$ is called a two level hierarchical structure of $\mathcal{M}$, where ψ is partition function $\psi(x) = \overline{X}_i$ if $x \epsilon X_i$. $\mathcal{M}^\pi$ is called *high level system* in this structure. Each of $\mathcal{M}_i$'s is called *low level system.* ψ is called *communication function.*

3 Syntax of Hierarchical COCOLOG

Beginning in this section we choose an element $\pi \epsilon IBCP(\mathcal{M})$ and the associated two level hierarchical structure to describe the hierarchical COCOLOG system. A hierarchical COCOLOG system is a family of first order theories $\{Th_k^\pi, Th_k^1. \cdots, Th_k^{|\pi|}, Th_k^c;\ k \geq 0\}$. Each of these theories has its own syntax, semantics and inference rules, wherein each represents the properties of one component in the hierarchical structure. Each one may take its own extra-logical transition to a subsequent theory at the same level through the extra-logical control rules of that level. Each of them has the facility to communicate with theories at the other level through another extra-logical rule called instantiation rule or block-position rule. The full detail of COCOLOG can be found in [CW92]. In order to define a two level hierarchical COCOLOG control structure for $\mathcal{M}$, we extend the basic COCOLOG languages [CW92] L_k by the addition of (i) a high level COCOLOG language L_k^π, (ii) a inter-layer communication language L^c, and (iii) attach one sort language for each of components in the structure. These are described briefly below.

The COCOLOG Language of a Hierarchical Structure $k \geq 0$:

$$L_k = L_k^\pi \cup L_k^c \cup L_k^1 \cup L_k^2 \cup \cdots \cup L_k^{|\pi|}.$$

where L_k^π is a *block-sort COCOLOG language* for $\mathcal{M}^\pi$, L_k^i is *$i-$sort COCOLOG languages* for $\mathcal{M}_i$, L_k^c is called *communication sort language.* The sets of constant symbols, function symbols and predicate symbols are list below for each sort language are listed below:
For $L_k^\pi, k \geq 0$:

$$Const(L_k^\pi) = \{\overline{X}_1, \cdots, \overline{X}_p, \overline{U}_i^j, \cdots, \overline{U}_k^l, \overline{U(0)}, \cdots \overline{U(k-1)}, \overline{0}, \overline{1}, \cdots, \overline{K(p)+1}\};$$

$$Func(L_k^\pi) = \{\Phi^\pi, +^\pi, -^\pi\}, \quad Pred(L_k^\pi) = \{Rbl^\pi(\cdot,\cdot,\cdot), Eq^\pi(\cdot,\cdot), CSE_k^\pi()\}.$$

For $1 \leq i \leq p$,

$$Const(L_k^i) = X_i \cup X_i^e \cup U \cup I_{K(n_i)}, \qquad Func(L_k^i) = \{\Phi^i, +, -\},$$

$$Pred(L_k^i) = \{Rbl^i(\cdot,\cdot,\cdot), Eq^i(\cdot,\cdot), CSE_k^i(), E_j^i(),\ 1 \leq j \leq |\pi| \& j \neq i\},$$

where the elements of X_e^i are called entry states from block X_i and $X_i \cap X_e^i = \emptyset$, $\Phi_i : X_i \times U_i \rightarrow X_i \cup X_e^i$. $E_j^i()$ is the new predicate called entry state predicate from X_i to X_j.
For L_k^c:

$$Const(L_k^c) = \{\overline{X}_1, \overline{X}_2, \cdots \overline{X}_p\} \cup X \cup U, Func(L_k^c) = \{\psi(\cdot,\cdot)\}, \ Pred(L_k^c) = \{Eq^c(\cdot,\cdot)\},$$

where $\psi : X \rightarrow \pi$. $WFF(L_k^\pi)$, $WFF(L_k^i)$ and $WFF(L_k^c)$ are the sets of well formed formula by the Backus-Naur rules with respect to L_k^π, L_k^i and L_k^c respectively.

4 Axiomatization and Proof Theory of a Hierarchical COCOLOG

The axiom set of a hierarchical COCOLOG consists of several parts. Each of them lies in one sorted language. and inference in each component at any given level will be carried out only within the corresponding of this language and axiom set.

Definition 4.1 The axiom set Σ_k^π of the high level COCOLOG theory Th_k^π consists of the following:
(1) Block transition axiom set: $AXM^{dyn}(L_0^\pi)\underline{\Delta}\{Eq_\pi(\Phi^\pi(\overline{X}^i,\overline{U}_i^j),\overline{X}^j);\ 1\leq i,j\leq p\}$; (2) Block reachability axiom set: $AXM^{Rbl}(L_0^\pi)$; (3) Arithmetic function axiom set $AXM^{arith}(L_k^\pi)$; (4) Equality axiom set $AXM^{Eq}(L_k^\pi)$; (5) Logical axiom set $AXM^{log}(L_k^\pi)$; (6) Size axiom set $AXM^{size}(L_k^\pi)$; (6) State estimation axiom set $AXM^{est}(L_k^\pi)$ when $k>0$.

The reader should refer to [1] for full definitions of the above axiom sets. The inference rules are simply the restricted *modus ponens* and *generalization* :

Definition 4.2 Set of Inference Rules $IR_k^\pi=\{MP(L^\pi,G(L^\pi)\}$, where

$$MP(L_k^\pi):\ \frac{A,A\rightarrow B}{B},\quad A,B\in WFF(L_k^\pi),\qquad G:\frac{A}{\forall xA},\quad A\in WFF(L_{\pi,k}),x\in L_k^\pi,$$

provided x does not occur free in A for rule G.

With the axiom set Σ_k^π, one can generate a new formula A called a *theorems* by applying the rules of inference to the axioms in a systematic way. In this case, we say A is *provable from* Σ_k^π and this is denoted by $\Sigma_k^\pi\vdash A$. At any instant k, the collection of all such theorems is called a *high level theory*, and this is denoted

$$Th_k^\pi\underline{\Delta}\{A;\ A\in WFF(L_k^\pi)\&\Sigma_k^\pi\vdash_{IR_k^\pi}A\}.$$

This set includes all truths about $\mathcal{M}^\pi$ is expressible in terms of the first-order language L_k^π.

Definition 4.3 The axiom set Σ_k^i of the low level COCOLOG theory Th_k^i consists of the following:
(1) Transition axiom set for $\mathcal{M}_i$: $AXM^{dyn}(L_0^i)\underline{\Delta}\{Eq^i(\Phi^i(x^i,u^q),x^{iq});\ x^i\in X_i\&u^q\in U\&x^{iq}\in X_i\cup X_i^e\}$. (2) Local reachability axiom set: $AXM^{Rbl^i}(L_k^i)$; (3) $AXM^{arith}(L_k^i)$ (4) $AXM^{Eq}(L_k^i)$; (5) $AXM^{log}(L_k^i)$; (6) Size axiom set $AXM^{size}(L_k^i)$; (7) State estimation axiom set: $AXM^{est}(L_k^i)$ when $k>0$; (8)] Exit state axiom set: $AXM^{exit}(L_0^i)\underline{\Delta}\{E_j^i(x');\ j\neq i\&\ x'\in X_e^i\}$, whenever $\exists x_i\in X_i,u\in U(\Phi(x_i,u)=x')$,

Definition 4.4 Inference Rules $IR(L_k^i)=\{MP(L_k^i),G(L_k^i)\}$. These correspond to these in Definition 4.2.

Similarly, the following set of formulas is called *low level theory for* $\mathcal{M}_i$:

$$Th_k^i\underline{\Delta}\{A;\ A\in WFF(L_k^i)\&\Sigma_k^i\vdash_{IR_k^i}A\}.$$

Th_k^c can be defined in a similar way. We only remark here that

$$\Sigma^c=\{Eq^c(\psi(x^i),\overline{X}_i);\ x^i\in X_i,\forall x\in X\wedge\overline{X}_i\in\pi\}\cup AXM^{Eq^c}\cup AXM^{log}(L_k)\cup AXM^{size}(L_k^c).$$

Theorem 4.1 For $k \geq 0, 1 \leq i \leq |\pi|$, Th_k^{π}, Th_k^i , Th_k^c are consistent and decidable.

The following theorem shows that the set of unrestricted inference rules $IR_k = \{MP(L_k), G(L_k)\}$ does not provide more theorems than IR_k^i with respect to $WFF(L_k^i)$.

Theorem 4.2 For $F \in WFF(L_k^i)$,

$$\Sigma_k^i \vdash_{IR(L_k^i)} F \Longleftrightarrow \Sigma_k^i \vdash_{IR(L_k)} F.$$

In terms of semantics, we remark that one should define the model of a hierarchical COCOLOG theory in such way that the algebraic property of the domain reflect the concepts listed in the previous Section 1.

5 Extra-logical Transitions in Hierarchical COCOLOG

For each COCOLOG theory, there is one set of extra-logical rules called *extra-logical control rules*, these make an extra-logical transition from one theory to another one at one given same level. In a hierarchical COCOLOG system, each theory has at least one additional (maybe two additional in the case of more layers involved), which pass the information from one level to another. We call one that passes information form high level system to low level system *instantiation rule*, which assign a control objective to low level system. The intuition behind this rule is to tell low level system what to do. We call one that passes information from low level system to high level system *block membership rule*, which reports to high level system which block the low level system is in. The construction of the following rules is depend upon the task. We assume that set of tasks is to drive the base system from the current state to the target state x^T.
High level control rule. For each $\overline{U}_i^j \in \mathcal{U}$:

$$\textbf{if} \quad F_i^j(L_k^{\pi}) \quad \textbf{then} \quad Eq^{\pi}(\overline{U(k)}, \overline{U}_i^j). \tag{3}$$

Low level control rule for $\mathcal{M}_i$: For $u^p \in U_i$

$$\textbf{if} \quad D_p(L_k^i)(x) \quad \textbf{then} \quad Eq^i(U_i(k), u^p), \tag{4}$$

where x appears free in $D_p(L_k^i)$, which will be instantiated by a state constant (current target state) according the following rules.
Instantiation Rule INS_k for $\overline{U}_i^j \in \mathcal{U}$

$$\textbf{IF} \quad Eq^{\pi}(\overline{U(k)}, \overline{U}_i^j) \quad \textbf{THEN} \quad \forall x, (E_j^i(x) \rightarrow D_p(L_k^i)(x/x^j)), \quad i \neq j. \tag{5}$$

$$\textbf{IF} \quad Eq^{\pi}(\overline{U(k)}, \overline{U}_i^i) \quad \textbf{THEN} \quad Eq^i(x_i^T, x^T), \quad i = j. \tag{6}$$

Notice **THEN** part lies in $WF(L_k^i)$, so it can be verified from Σ_k^i.

Then the submachine $\mathcal{M}_i$ is invoked and Th_k^i takes its theory transition through extra-logic control rule:

Usually, it takes several low level control steps to achieve this control objective. The following rule reports back to high level when it has been achieved.

Block-membership Rule BM_k^i: For each $x' \in X_e^i$:

$$\textbf{IF} \quad CSE_k^i(x_i^T) \quad \textbf{THEN} \quad CSE_{k+1}^{\pi}(\psi(x')). \tag{7}$$

This rule does not provided any information to high level system before the current target state is reached, because there is no new information in terms of block position. Once the control objective has been achieved by low level system, the block position is changed, then the high level COCOLOG theory take one theory transition.

Example 5.1 See Figure 1-(a). Suppose that the current and target states of $\mathcal{M}_8$ at time $k=0$ are x^3 and x^4 respectively. So the current and target states of high level system $\mathcal{M}^{\pi}$ are X_1 and X_2 respectively. Suppose in (3), $F_1^2(L_k^{\pi}) = \exists \overline{X}' \exists l, CSE^{\pi}(\overline{X}') \wedge Rbl^{\pi}(\overline{X}', \overline{X}_2, \bar{l})$, then either $\overline{U}_1^2$ or $\overline{U}_1^3$ may be invoked. if $Eq^{\pi}(\overline{U(1)}, \overline{U}_1^2)$, then from (5) the task for $\mathcal{M}_1$ is specified by the formula $Eq^1(x_1^T, x^5)$. $D_p(L_k^1)(x/x^5) = \exists l, \exists x, CSE_k^1(x) \wedge Rbl^1(x, x^5, l)$. Then by (4) Th_1^1 decide $Eq^1(U_1(1), b)$ and switches to Th_2^1 and so on. Finally Th_3^1 decides $Eq^1(U_1(3), b)$ and the current state is x^5. Since $\psi(x^5) = \overline{X}_2$, by (7) Th_1^{π} switches to Th_2^{π} and repeat the above operation for $\mathcal{M}_2$ until x^4 is reached.

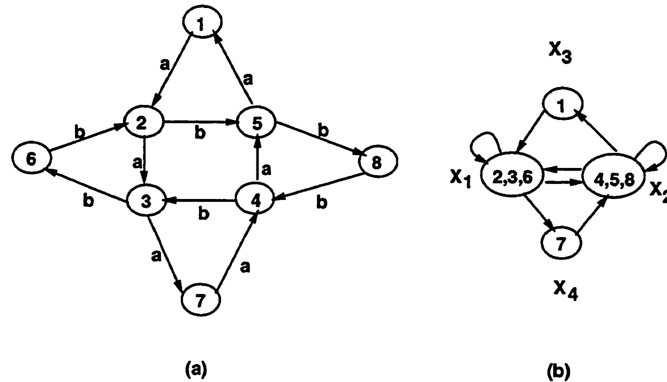


Figure 1: (a) A 8 state machine $\mathcal{M}_8$ and one partition π, (b) Extra-logical transition of hierarchical COCOLOG system.

REFERENCES

[CW92] P. E. Caines, S.Wang, COCOLOG: A conditional observer and controller logic for finite machines, under revision for SICOPT.

[CWei94] P.E. Caines. Y.J.Wei The hierarchical lattice of a finite machine, submitted for publication to *System and Control Letters*, Feb, 1994

[G68] A. Ginzburg, *Algebraic Theory of Automaton*, Academic Press, New York, 1968

[HS66] J. Hartmanis, R.E. Stearns, *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[LW90] F. Lin, W.M. Wonham, Decentralized control and coordination of discrete-event systems with partial observation, *IEEE Trans. on Automatic Control*, V. 35, Dec. 1990.

[RW92] K. Rudie, W.M. Wonham Think globally, act locally: decentralized supervisory control, *IEEE Trans. on Automatic Control*, V. 37, Nov. 1992.
[ZW] H. Zhong and W.M. Wonham, On the consistency of hierarchical supervision in discrete-event systems, *IEEE Trans. on Automatic Control,* 35(10) Oct. 1990.

Une approche relationnelle à la décomposition parallèle

Brahim Chaib-draa, Jules Desharnais, Ridha Khédri,
Imed Jarras, Slim Sayadi, Fairouz Tchier
Département d'informatique, Université Laval
Québec, QC, G1K 7P4 Canada

1 Introduction

Schmidt et Ströhlein [10, 11] définissent de manière relationnelle un concept qu'ils appellent *diagramme de programme.* Ils utilisent ensuite ce concept pour l'étude des programmes séquentiels. Nous partons de cette notion de diagramme de programme pour définir la sémantique d'un opérateur de composition parallèle (section 3). Bien que la définition s'applique de manière générale aux programmes avec des variables, nous nous restreignons, à la section 4, au cas des systèmes de transitions afin d'illustrer comment l'opérateur décrit leur produit synchronisé [1]. Finalement, à la section 5, nous présentons un exemple de résolution de l'équation $Q \parallel X = P$ (décomposition parallèle de P).

La prochaine section débute par une exposition de l'axiomatique des algèbres de relations. Mais par la suite, nous utilisons le modèle usuel de telles algèbres (celui des relations sur des ensembles), afin d'éviter un traitement complètement axiomatique qui allongerait indûment cet article.

2 Algèbres de relations

L'origine du calcul des relations remonte au siècle dernier avec les travaux de De Morgan, Peirce, Dedekind et Schröder. Leur étude a été ravivée par les articles de Chin et Tarski [4, 12], qui axiomatisent la notion d'algèbre de relations homogènes. La définition suivante est tirée de [11].

2.1 Définition. Une *algèbre de relations homogène* est une structure $(\mathcal{R}, \cup, \cap, ^-, \circ, ^\smile)$ sur un ensemble $\mathcal{R}$ non vide dont les éléments sont appelés *relations.* Les conditions suivantes sont satisfaites :

1. $(\mathcal{R}, \cup, \cap, ^-)$ est une algèbre booléenne complète, avec élément *zéro* $\emptyset$ et élément *universel* L. Les éléments de $\mathcal{R}$ sont ordonnés par *inclusion*, notée $\subseteq$.

2. La composition est associative et a pour identité I : $P \circ (Q \circ R) = (P \circ Q) \circ R$ et $I \circ R = R \circ I = R$.

3. $P \circ Q \subseteq R \Leftrightarrow \widehat{P} \circ \overline{R} \subseteq \overline{Q} \Leftrightarrow \overline{R} \circ \widehat{Q} \subseteq \overline{P}$ (*règle de Schröder*).

4. Si $R \neq \emptyset$, alors $L \circ R \circ L = L$ (*règle de Tarski*). □

Le modèle usuel de ces axiomes est celui des relations sur un ensemble. Dans ce modèle, les opérations d'union ($\cup$), d'intersection ($\cap$) et de complémentation ($^-$) sont les opérations ensemblistes habituelles, l'inverse d'une relation est $\widehat{R} = \{(x,y) \mid (y,x) \in R\}$ et la composition est donnée par $Q \circ R = \{(x,z) \mid \exists y : (x,y) \in Q \wedge (y,z) \in R\}$. Nous utilisons ci-dessous des algèbres hétérogènes, dont le modèle usuel est celui de relations entre ensembles différents. Leur définition est essentiellement semblable à la définition 2.1 (voir [11]); toutefois, il faut introduire la notion de type et d'opérations partielles. Ainsi, si $Q, R \subseteq S \times T$, alors les opérations $Q \cup R$ et $Q \cap R$ sont définies. De même, si $Q \subseteq S \times T$ et $R \subseteq T \times U$, alors la composition $Q \circ R$ est définie. Quand il y a plusieurs ensembles impliqués, il peut y avoir plusieurs relations relations universelles, zéros et identités (par exemple, $I \subseteq S \times S$ et $I \subseteq T \times T$); par simplicité, elles sont toutes dénotées par $L, \emptyset, I$, respectivement. La précédence des opérateurs relationnels, de la plus élevée à la plus faible, est la suivante : $^-$ et $\widehat{}$ ont la même priorité, suivis par $\circ$, suivi par $\cap$ et finalement par $\cup$. Par la suite, nous omettons le symbole de composition $\circ$ et nous écrivons simplement QR pour $Q \circ R$. De plus, nous écrivons $(R)\widehat{}$ plutôt que $\widehat{(R)}$ pour les expressions munies de parenthèses.

De cette définition, on dérive les règles habituelles du calcul des relations (voir, par exemple, [4, 11]). Nous supposons ces règles connues et nous en rappelons quelques-unes, incluant quelques lois booléennes.

2.2 Soient P, Q, R des relations. Alors,

1. $\overline{Q \cup R} = \overline{Q} \cap \overline{R}$,
2. $\overline{Q \cap R} = \overline{Q} \cup \overline{R}$,
3. $\overline{\overline{R}} = R$,
4. $P \cap Q \subseteq R \Leftrightarrow P \subseteq \overline{Q} \cup R$,
5. $Q \subset R \;\Leftrightarrow\; \overline{R} \subset \overline{Q}$,
6. $P(Q \cap R) \subset PQ \cap PR$,
7. $(P \cap Q)R \subset PR \cap QR$,
8. $P(Q \cup R) = PQ \cup PR$,
9. $(P \cup Q)R = PR \cup QR$,
10. $Q \subset R \;\Rightarrow\; PQ \subset PR$,
11. $P \subset Q \;\Rightarrow\; PR \subset QR$,
12. $Q \subset R \;\Leftrightarrow\; \widehat{Q} \subset \widehat{R}$,
13. $(Q \cup R)\widehat{} = \widehat{Q} \cup \widehat{R}$,
14. $(Q \cap R)\widehat{} = \widehat{Q} \cap \widehat{R}$,
15. $(QR)\widehat{} = \widehat{R}\widehat{Q}$,
16. $\widehat{\widehat{R}} = R$,
17. $\widehat{\overline{R}} = \overline{\widehat{R}}$.

2.3 Définition. Une relation R est *déterministe* ssi $\widehat{R}R \subseteq I$; elle est *totale* ssi $L = RL$ (ou encore, $I \subseteq R\widehat{R}$); elle est *injective* ssi $\widehat{R}$ est déterministe (*i.e.* $R\widehat{R} \subseteq I$); elle est *surjective* ssi $\widehat{R}$ est totale (*i.e.* $LR = L$, ou $I \subseteq \widehat{R}R$). □

2.4 Soient P, Q et R des relations. Alors,

1. P déterministe $\Rightarrow P(Q \cap R) = PQ \cap PR$,
   P injective $\Rightarrow (Q \cap R)P = QP \cap RP$;
2. P déterministe $\Rightarrow (Q \cap R\widehat{P})P = QP \cap R$,
   P injective $\Rightarrow P(\widehat{P}Q \cap R) = Q \cap PR$.

2.5 Définition. Un n-uplet de relations $(\pi_1, \ldots, \pi_n)$ est un *produit direct* ssi

$$\widehat{\pi}_i\pi_i = I \quad (i = 1 \ldots n), \qquad \bigcap_{i=1}^{n} \pi_i\widehat{\pi}_i = I.$$

On dit que le produit est *plein* si $i \neq j \Rightarrow \widehat{\pi}_i\pi_j = L \quad (i, j = 1 \ldots n)$. Les relations π_i sont appelées *projections.* □

Cette définition implique, entre autres, qu'une projection est une relation totale, déterministe et surjective. Par exemple, soient les ensembles S, T et U. On vérifie facilement que les projections $\pi_1 : S \times T \times U \to S \times T$ et $\pi_2 : S \times T \times U \to T \times U$ forment un produit direct (notons que $\pi_1 = \{((s,t,u),(s,t)) \mid s \in S \wedge t \in T \wedge u \in U\}$ et $\pi_2 = \{((s,t,u),(t,u)) \mid s \in S \wedge t \in T \wedge u \in U\}$). De même, les projections $\pi_S : S \times T \times U \to S$, $\pi_T : S \times T \times U \to T$ et $\pi_U : S \times T \times U \to U$ forment un plein produit direct. La définition de plein produit direct donnée ci-dessus est appelée *produit direct* dans [11]; nous avons tiré de [7] la distinction entre le produit direct et le plein produit direct.

2.6 Proposition. *Soit $(\pi_1, \ldots, \pi_n)$ un plein produit direct et soient $R_i \neq \emptyset \quad (i = 1 \ldots n)$ des relations. Ces relations satisfont la loi $\widehat{\pi}_i(\bigcap_{j=1}^{n} \pi_j R_j \widehat{\pi}_j)\pi_i = R_i \quad (i = 1 \ldots n)$.* □

Pour des relations sur des ensembles et un plein produit direct (π_1, π_2), cette proposition est facile à vérifier en remarquant que

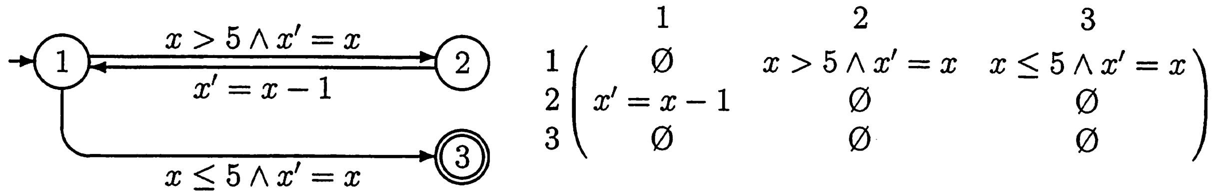
$$\pi_1 R_1 \widehat{\pi}_1 \cap \pi_2 R_2 \widehat{\pi}_2 = \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in R_1 \wedge (x_2, y_2) \in R_2\}.$$

3 Diagrammes de programmes et composition parallèle

Dans cette section, nous présentons la notion de diagramme de programme, puis nous définissons un opérateur de composition parallèle pour combiner deux diagrammes.

3.1 Définition. Soient les ensembles S (l'ensemble des *états*), et V (l'ensemble des *points de contrôle*). Un *diagramme de programme* sur S et V est une relation P sur $S \times V$ (*i.e.* $P \subseteq (S \times V) \times (S \times V)$). □

Donnons un exemple qui illustre cette définition et qui justifie par la même occasion le terme *diagramme de programme*. Considérons la boucle `tant que x>5 faire x:=x-1`, où la variable x prend ses valeurs sur l'ensemble des naturels $\mathbf{N}$, qui est l'ensemble des états (plus généralement, l'ensemble des états est le produit cartésien des ensembles sur lesquels les variables du programme prennent leur valeur). Un tel programme est conventionnellement représenté par le diagramme suivant sur l'ensemble des points de contrôle $\{1,2,3\}$:



Le sommet 1 est le point d'entrée et le sommet 3 est le point de sortie. Chaque arc est étiqueté par un prédicat qui décrit la relation calculée par le programme entre les sommets reliés par cet arc; ainsi, la relation calculée par le corps de boucle entre les points 2 et 1 est $\{(x,x') \mid x \in \mathbf{N} \wedge x' = x-1\}$. Un tel diagramme peut être représenté par la matrice adjacente au diagramme ou par la relation suivante sur $\mathbf{N} \times \{1,2,3\}$:

$$\begin{aligned}
&\{((x,1),(x',2)) \mid x > 5 \wedge x' = x\} \cup \\
&\{((x,1),(x',3)) \mid x \leq 5 \wedge x' = x\} \cup \\
&\{((x,2),(x',1)) \mid x' = x-1\}.
\end{aligned}$$

Dans la notation matricielle, une entrée $\emptyset$ correspond à l'absence d'arc entre deux sommets (ou encore à la présence d'un arc étiqueté par $\emptyset$). Notons qu'avec la représentation matricielle ou la représentation ensembliste nous avons perdu les notions de point d'entrée et de point de sortie. Comme nous n'en avons pas besoin dans cette présentation, nous n'indiquerons pas comment en tenir compte. Pour une définition relationnelle complète, axiomatique, de la notion de diagramme de programme, il faut consulter [10, 11]. Par la suite, nous adoptons un point de vue sémantique et nous utilisons simplement le terme *programme* pour *diagramme de programme*, bien qu'en toute rigueur, un programme soit le texte auquel est associé le diagramme. De plus, la correspondance entre les lignes ou colonnes des matrices et les points de contrôle sera omise (contrairement à l'exemple ci-dessus) lorsque le contexte permet de la déterminer.

Un ensemble de matrices de dimensions appropriées constitue une algèbre de relations [11], en définissant les opérateurs relationnels comme suit (R_{ij} désigne l'entrée i,j de la matrice R) :

3.2 $$(Q \cup R)_{ij} = Q_{ij} \cup R_{ij}, \qquad (\overline{R})_{ij} = \overline{R_{ij}}, \qquad (QR)_{ij} = \textstyle\bigcup_k Q_{ik}R_{kj},$$
$$(Q \cap R)_{ij} = Q_{ij} \cap R_{ij}, \qquad (\widehat{R})_{ij} = (R_{ji})\widehat{\ }.$$

Par exemple, considérons la composition

$$\begin{pmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} R_{11}R_{11} \cup R_{12}R_{21} & R_{11}R_{12} \cup R_{12}R_{22} \\ R_{21}R_{11} \cup R_{22}R_{21} & R_{21}R_{12} \cup R_{22}R_{22} \end{pmatrix}.$$

Remarquons comment le produit matriciel fait à la fois la composition usuelle des graphes et la composition des relations sur les chemins traversés. Ainsi, il y a deux chemins possibles pour passer du sommet 1 au sommet 2 par la composition ci-dessus :

- Du sommet 1 au sommet 1 par la première matrice et du sommet 1 au sommet 2 par la deuxième. La composition des relations sur ce chemin est $R_{11}R_{12}$.
- Du sommet 1 au sommet 2 par la première matrice et du sommet 2 au sommet 2 par la deuxième. La composition des relations sur ce chemin est $R_{12}R_{22}$.

Ceci explique le contenu de l'entrée (1,2) de la matrice résultant de la composition. Finalement, remarquons, à titre d'exemple, que pour les matrices 2×2, les relations zéro, universelle et identité sont respectivement $\begin{pmatrix} \emptyset & \emptyset \\ \emptyset & \emptyset \end{pmatrix}$, $\begin{pmatrix} L & L \\ L & L \end{pmatrix}$ et $\begin{pmatrix} I & \emptyset \\ \emptyset & I \end{pmatrix}$, ce qui conduit à écrire des expressions comme $L = \begin{pmatrix} L & L \\ L & L \end{pmatrix}$.

Nous sommes maintenant prêts à définir la composition parallèle de programmes.

3.3 Définition. Soient les programmes P_1 et P_2, tels que P_1 est une relation sur $T \times S_1 \times V_1$ et P_2 est une relation sur $T \times S_2 \times V_2$ (l'ensemble des états de P_i est $T \times S_i$ et l'ensemble de ses points de contrôle est V_i, pour $i = 1, 2$). Les composantes S_1 et S_2 sont les composantes propres à P_1 et P_2, respectivement, alors que T est une composante partagée. Soit le produit direct (π_1, π_2), où π_1 et π_2 sont les projections

$$\begin{aligned} &\pi_1 : T \times S_1 \times S_2 \times V_1 \times V_2 \to T \times S_1 \times V_1, \\ &\pi_2 : T \times S_1 \times S_2 \times V_1 \times V_2 \to T \times S_2 \times V_2. \end{aligned}$$

La *composition parallèle* de P_1 et P_2, notée $P_1 \parallel P_2$, est la relation sur $T \times S_1 \times S_2 \times V_1 \times V_2$ définie par

$$P_1 \parallel P_2 \stackrel{\text{def}}{=} \pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 P_2 \widehat{\pi}_2 \cup \pi_1 \widehat{\pi}_1 \cap \pi_2 P_2 \widehat{\pi}_2 \cup \pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 \widehat{\pi}_2. \qquad \square$$

Afin d'expliquer cette définition, supposons que P_1 et P_2 soient définis par les prédicats p_1 et p_2 :

$$\begin{aligned} &P_1 = \{((t, s_1, v_1), (t', s_1', v_1')) \mid p_1(t, s_1, v_1, t', s_1', v_1')\}, \\ &P_2 = \{((t, s_2, v_2), (t', s_2', v_2')) \mid p_2(t, s_2, v_2, t', s_2', v_2')\}. \end{aligned}$$

Il est facile de vérifier que

$$\begin{aligned} &\pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 P_2 \widehat{\pi}_2 \\ &= \{((t, s_1, s_2, v_1, v_2), (t', s_1', s_2', v_1', v_2')) \mid p_1(t, s_1, v_1, t', s_1', v_1') \wedge p_2(t, s_2, v_2, t', s_2', v_2')\}. \end{aligned}$$

Le premier terme de la définition de $P_1 \parallel P_2$ représente donc les transitions simultanées de P_1 et P_2. Ces transitions sont possibles à la condition que P_1 et P_2 traitent la composante commune T de la même manière; en effet, les conditions imposées à la composante T sont définies par la conjonction de p_1 et p_2. De même, on montre que

$$\pi_1\widehat{\pi}_1 \cap \pi_2 P_2 \widehat{\pi}_2$$
$$= \{((t, s_1, s_2, v_1, v_2), (t', s_1', s_2', v_1', v_2')) | t' = t \wedge s_1' = s_1 \wedge v_1' = v_1 \wedge p_2(t, s_2, v_2, t', s_2', v_2')\}.$$

Le deuxième terme de $P_1 \parallel P_2$ est donc l'ensemble des transitions effectuées par P_2 alors que P_1 est inactif (ou fait la transition identité I). Ces transitions sont possibles à la condition que le programme P_2 ne modifie pas la valeur de la composante T. Le troisième terme de $P_1 \parallel P_2$ a une interprétation symétrique.

Les lois 2.2(8,9) nous permettent de donner une expression alternative pour $P_1 \parallel P_2$:

3.4 $P_1 \parallel P_2 = \pi_1(P_1 \cup I)\widehat{\pi}_1 \cap \pi_2 P_2 \widehat{\pi}_2 \cup \pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 \widehat{\pi}_2$;

c'est cette formulation que nous utiliserons par la suite.

Notons que la définition 3.3 se généralise facilement au cas où l'une (ou plusieurs) des composantes T, S_1, S_2 est absente.

La prochaine section donne un exemple concret de produit parallèle. On y montre aussi comment les projections utilisées dans la définition du produit parallèle peuvent être représentées par des matrices, comme le sont les programmes.

4 Composition parallèle de systèmes de transitions

Dans cette section, nous montrons comment associer une relation à un système de transitions de sorte que celui-ci soit vu comme un programme simple; puis nous donnons un exemple de composition parallèle de systèmes de transitions.

Soient les systèmes de transitions P_1 et P_2 suivants (nous omettons de désigner un état initial) :

Afin de considérer ces systèmes de transitions comme des diagrammes de programmes, il suffit d'indiquer quelles sont les relations qui correspondent aux événements a, b et c. Une première approche serait de considérer que tous ces événements sont des relations sur un même ensemble S. Nous adoptons une approche différente et nous considérons que a, b et c sont des relations sur des ensembles A, B et C, respectivement, de sorte que P_1 et P_2 sont des relations sur $A \times B \times V_1$ et $A \times C \times V_2$, respectivement, avec $V_1 = V_2 = \{1, 2\}$. Ceci permet, d'une part, de définir la notion de projection sur certains événements en utilisant des projections conventionnelles (qui éliminent des composantes cartésiennes) et, d'autre part, d'illustrer la définition 3.3 dans toute sa généralité, puisque P_1 a une composante propre (B), ainsi que P_2 (la composante C) et qu'ils partagent la composante A. Les seules contraintes que nous imposons ici à a, b et c sont $a \cap I = \emptyset, b \cap I = \emptyset, c \cap I = \emptyset, a \neq \emptyset, b \neq \emptyset$ et $c \neq \emptyset$; les trois premières contraintes signifient que a, b et c ne peuvent jamais être confondues avec l'identité I. Par exemple, nous pourrions choisir $A = B = C = \{0, 1\}$ et $a = b = c = \overline{I}$.

Les relations P_1 et P_2 sont les suivantes :

$$P_1 = \begin{pmatrix} \emptyset & \underline{aI} \\ \underline{Ib} & \emptyset \end{pmatrix}_{AB}, \qquad P_2 = \begin{pmatrix} \emptyset & \underline{Ic} \\ \underline{aI} & \emptyset \end{pmatrix}_{AC}.$$

Les entrées de la matrice P_1 sont des relations sur $A \times B$ et celles de P_2 des relations sur $A \times C$ (c'est ce que rappellent les indices de ces matrices). Soient les projections $\pi_{A_1} : A \times B \to A$ et $\pi_{B_1} : A \times B \to B$. La notation $\underline{xy}$ est une abréviation qui vise à limiter la taille des matrices; cette abréviation est définie par $\underline{xy} \stackrel{\text{def}}{=} \pi_{A_1} x \widehat{\pi}_{A_1} \cap \pi_{B_1} y \widehat{\pi}_{B_1}$. Par exemple, l'entrée $\underline{aI}$, dans la matrice P_1, est la relation sur $A \times B$ telle que la composante A change selon la relation a, alors que la composante B ne change pas; dans le cas de l'entrée $\underline{aI}$ de la matrice P_2, c'est la composante C qui ne change pas (pour P_2, la relation $\underline{aI}$ est construite avec un produit direct différent). Les relations $\underline{aI}$ et $\underline{Ib}$ de la matrice P_1 (par exemple) satisfont les lois suivantes (voir proposition 2.6) :

$$\textbf{4.1} \quad \begin{array}{llll} \widehat{\pi}_{A_1} \underline{aI}\, \pi_{A_1} = a, & \widehat{\pi}_{A_1} \underline{Ib}\, \pi_{A_1} = I, & \widehat{\pi}_{B_1} \underline{aI}\, \pi_{B_1} = I, & \widehat{\pi}_{B_1} \underline{Ib}\, \pi_{B_1} = b, \\ \pi_{A_1} a \widehat{\pi}_{A_1} = \underline{aL}, & \pi_{A_1} I \widehat{\pi}_{A_1} = \underline{IL}, & \pi_{B_1} I \widehat{\pi}_{B_1} = \underline{LI}, & \pi_{B_1} b \widehat{\pi}_{B_1} = \underline{Lb}. \end{array}$$

4.2 Remarque. Ceci se généralise facilement à des projections différentes ou aux cas où le nombre de composantes des entrées des matrices est différent de deux.

Pour l'exemple traité ici, les relations π_1 et π_2 de la définition 3.3 ont comme fonctionnalité

$$\begin{array}{l} \pi_1 : A \times B \times C \times V_1 \times V_2 \to A \times B \times V_1, \\ \pi_2 : A \times B \times C \times V_1 \times V_2 \to A \times C \times V_2. \end{array}$$

Ces projections peuvent se représenter par des matrices, tout comme les diagrammes de programmes. Une paire π_1, π_2 convenable est la paire suivante :

$$\textbf{4.3} \quad \pi_1 = \begin{array}{c} \\ 11 \\ 12 \\ 21 \\ 22 \end{array} \!\!\begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{pmatrix} \omega_1 & \emptyset \\ \omega_1 & \emptyset \\ \emptyset & \omega_1 \\ \emptyset & \omega_1 \end{pmatrix} \end{array}, \qquad \pi_2 = \begin{array}{c} \\ 11 \\ 12 \\ 21 \\ 22 \end{array} \!\!\begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{pmatrix} \omega_2 & \emptyset \\ \emptyset & \omega_2 \\ \omega_2 & \emptyset \\ \emptyset & \omega_2 \end{pmatrix} \end{array},$$

où $\omega_1 : A \times B \times C \to A \times B$ et $\omega_2 : A \times B \times C \to A \times C$ sont des projections qui n'impliquent que les ensembles d'états. Ces matrices sont déterminées à une permutation près seulement, puisque l'ordre des étiquettes des lignes et des colonnes est arbitraire. La structure des matrices détermine les projections de $V_1 \times V_2$ sur V_1 ou V_2; par exemple, π_1 projette la paire (1,2) sur 1 alors que π_2 la projette sur 2. Les projections relatives aux ensembles d'états sont faites par ω_1 et ω_2. La paire (π_1, π_2) est un produit direct (définition 2.5). Vérifions par exemple la propriété $\widehat{\pi}_1 \pi_1 = I$; les autres propriétés se vérifient de la même manière.

$$\widehat{\pi}_1 \pi_1 = \begin{pmatrix} \widehat{\omega}_1 & \widehat{\omega}_1 & \emptyset & \emptyset \\ \emptyset & \emptyset & \widehat{\omega}_1 & \widehat{\omega}_1 \end{pmatrix} \begin{pmatrix} \omega_1 & \emptyset \\ \omega_1 & \emptyset \\ \emptyset & \omega_1 \\ \emptyset & \omega_1 \end{pmatrix} = \begin{pmatrix} \widehat{\omega}_1 \omega_1 & \emptyset \\ \emptyset & \widehat{\omega}_1 \omega_1 \end{pmatrix} = \begin{pmatrix} I & \emptyset \\ \emptyset & I \end{pmatrix} = I,$$

où nous avons utilisé 3.2 et le fait que la projection ω_1 satisfait $\widehat{\omega}_1 \omega_1 = I$ (définition 2.5).

Une projection comme π_1, qui élimine deux composantes, peut être exprimée de deux façons comme la composition de deux projections qui éliminent chacune une composante. En effet,

$$\textbf{4.4}\quad \pi_1 = \begin{pmatrix} \omega_1 & \emptyset \\ \omega_1 & \emptyset \\ \emptyset & \omega_1 \\ \emptyset & \omega_1 \end{pmatrix} = \begin{pmatrix} \omega_1 & \emptyset & \emptyset & \emptyset \\ \emptyset & \omega_1 & \emptyset & \emptyset \\ \emptyset & \emptyset & \omega_1 & \emptyset \\ \emptyset & \emptyset & \emptyset & \omega_1 \end{pmatrix} \begin{pmatrix} I & \emptyset \\ I & \emptyset \\ \emptyset & I \\ \emptyset & I \end{pmatrix}$$

$$= \Omega_1 \rho_1 = \begin{pmatrix} I & \emptyset \\ I & \emptyset \\ \emptyset & I \\ \emptyset & I \end{pmatrix} \begin{pmatrix} \omega_1 & \emptyset \\ \emptyset & \omega_1 \end{pmatrix} = \rho_1' \Omega_1'$$

(on vérifie facilement que les compositions de ces matrices donnent bien π_1). Les fonctionnalités de $\Omega_1, \rho_1, \rho_1'$ et Ω_1' sont

$$\begin{aligned} &\Omega_1 : A \times B \times C \times V_1 \times V_2 \to A \times B \times V_1 \times V_2, \\ &\rho_1 : A \times B \times V_1 \times V_2 \to A \times B \times V_1, \\ &\rho_1' : A \times B \times C \times V_1 \times V_2 \to A \times B \times C \times V_1, \\ &\Omega_1' : A \times B \times C \times V_1 \to A \times B \times V_1. \end{aligned}$$

Calculons les quatre sous-expressions de la définition de $P_1 \parallel P_2$ (équation 3.4). Tout d'abord, par les lois 3.2, 4.1 et la remarque 4.2,

$$\pi_1(P_1 \cup I)\widehat{\pi}_1 = \begin{pmatrix} \omega_1 & \emptyset \\ \omega_1 & \emptyset \\ \emptyset & \omega_1 \\ \emptyset & \omega_1 \end{pmatrix} \begin{pmatrix} \underline{II} & \underline{aI} \\ \underline{Ib} & \underline{II} \end{pmatrix}_{AB} \begin{pmatrix} \widehat{\omega}_1 & \widehat{\omega}_1 & \emptyset & \emptyset \\ \emptyset & \emptyset & \widehat{\omega}_1 & \widehat{\omega}_1 \end{pmatrix}$$

$$= \begin{pmatrix} \omega_1\underline{II}\widehat{\omega}_1 & \omega_1\underline{II}\widehat{\omega}_1 & \omega_1\underline{aI}\widehat{\omega}_1 & \omega_1\underline{aI}\widehat{\omega}_1 \\ \omega_1\underline{II}\widehat{\omega}_1 & \omega_1\underline{II}\widehat{\omega}_1 & \omega_1\underline{aI}\widehat{\omega}_1 & \omega_1\underline{aI}\widehat{\omega}_1 \\ \omega_1\underline{Ib}\widehat{\omega}_1 & \omega_1\underline{Ib}\widehat{\omega}_1 & \omega_1\underline{II}\widehat{\omega}_1 & \omega_1\underline{II}\widehat{\omega}_1 \\ \omega_1\underline{Ib}\widehat{\omega}_1 & \omega_1\underline{Ib}\widehat{\omega}_1 & \omega_1\underline{II}\widehat{\omega}_1 & \omega_1\underline{II}\widehat{\omega}_1 \end{pmatrix}_{ABC} = \begin{pmatrix} \underline{IIL} & \underline{IIL} & \underline{aIL} & \underline{aIL} \\ \underline{IIL} & \underline{IIL} & \underline{aIL} & \underline{aIL} \\ \underline{IbL} & \underline{IbL} & \underline{IIL} & \underline{IIL} \\ \underline{IbL} & \underline{IbL} & \underline{IIL} & \underline{IIL} \end{pmatrix}_{ABC}$$

(remarquez comment cette matrice est semblable à celle de $P_1 \cup I$; c'en est une sorte d'expansion dont les entrées indiquent que la relation sur la composante C est universelle). De même,

$$\pi_2 P_2 \widehat{\pi}_2 = \begin{pmatrix} \omega_2 & \emptyset \\ \emptyset & \omega_2 \\ \omega_2 & \emptyset \\ \emptyset & \omega_2 \end{pmatrix} \begin{pmatrix} \emptyset & \underline{Ic} \\ \underline{aI} & \emptyset \end{pmatrix} \begin{pmatrix} \widehat{\omega}_2 & \emptyset & \widehat{\omega}_2 & \emptyset \\ \emptyset & \widehat{\omega}_2 & \emptyset & \widehat{\omega}_2 \end{pmatrix} = \begin{pmatrix} \emptyset & \underline{ILc} & \emptyset & \underline{ILc} \\ \underline{aLI} & \emptyset & \underline{aLI} & \emptyset \\ \emptyset & \underline{ILc} & \emptyset & \underline{ILc} \\ \underline{aLI} & \emptyset & \underline{aLI} & \emptyset \end{pmatrix}_{ABC},$$

$$\pi_1 P_1 \widehat{\pi}_1 = \begin{pmatrix} \omega_1 & \emptyset \\ \omega_1 & \emptyset \\ \emptyset & \omega_1 \\ \emptyset & \omega_1 \end{pmatrix} \begin{pmatrix} \emptyset & \underline{aI} \\ \underline{Ib} & \emptyset \end{pmatrix} \begin{pmatrix} \widehat{\omega}_1 & \widehat{\omega}_1 & \emptyset & \emptyset \\ \emptyset & \emptyset & \widehat{\omega}_1 & \widehat{\omega}_1 \end{pmatrix} = \begin{pmatrix} \emptyset & \emptyset & \underline{aIL} & \underline{aIL} \\ \emptyset & \emptyset & \underline{aIL} & \underline{aIL} \\ \underline{IbL} & \underline{IbL} & \emptyset & \emptyset \\ \underline{IbL} & \underline{IbL} & \emptyset & \emptyset \end{pmatrix}_{ABC},$$

$$\pi_2 \widehat{\pi}_2 = \begin{pmatrix} \omega_2 & \emptyset \\ \emptyset & \omega_2 \\ \omega_2 & \emptyset \\ \emptyset & \omega_2 \end{pmatrix} \begin{pmatrix} \widehat{\omega}_2 & \emptyset & \widehat{\omega}_2 & \emptyset \\ \emptyset & \widehat{\omega}_2 & \emptyset & \widehat{\omega}_2 \end{pmatrix} = \begin{pmatrix} \underline{ILI} & \emptyset & \underline{ILI} & \emptyset \\ \emptyset & \underline{ILI} & \emptyset & \underline{ILI} \\ \underline{ILI} & \emptyset & \underline{ILI} & \emptyset \\ \emptyset & \underline{ILI} & \emptyset & \underline{ILI} \end{pmatrix}_{ABC}.$$

Il ne reste plus qu'à calculer $\pi_1(P_1 \cup I)\widehat{\pi}_1 \cap \pi_2 P_2 \widehat{\pi}_2 \cup \pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 \widehat{\pi}_2$. Le résultat est

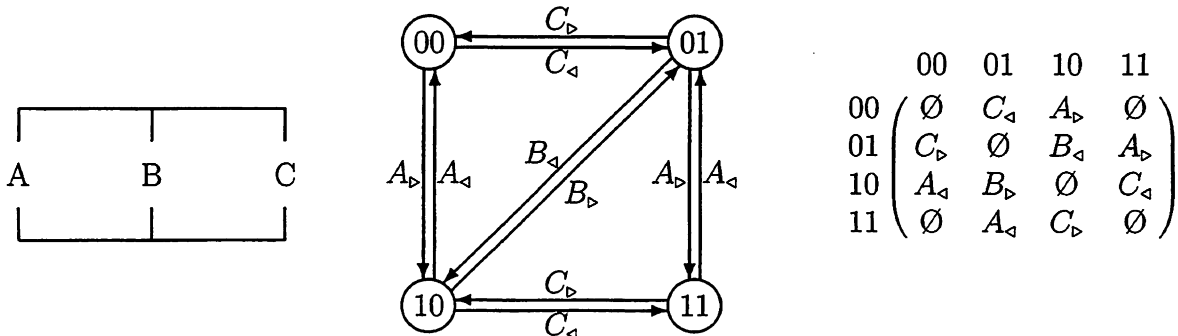
$$P_1 \parallel P_2 = \left(\begin{array}{cc|cc} \emptyset & \underline{IIc} & \emptyset & \emptyset \\ \emptyset & \emptyset & \underline{aII} & \emptyset \\ \hline \underline{IbI} & \underline{Ibc} & \emptyset & \underline{IIc} \\ \emptyset & \underline{IbI} & \emptyset & \emptyset \end{array}\right)_{ABC} .$$

$P_1 \parallel P_2$ est une relation sur $A \times B \times C \times V_1 \times V_2$. La matrice $P_1 \parallel P_2$ est divisée en quatre sous-matrices. La position d'une de ces sous-matrices correspond à une transition de P_1, alors que la position à l'intérieur d'une sous-matrice correspond à une transition de P_2. Par exemple, l'entrée $\underline{aII}$, qui est dans la sous-matrice qui occupe la position (1,2), correspond à une transition de P_1 du sommet 1 au sommet 2 (étiquetée a); la même entrée correspond à une transition de P_2 du sommet 2 au sommet 1 (aussi étiquetée a), puisqu'elle occupe la position (2,1) dans la sous-matrice. Ceci correspond à une transition simultanée de P_1 et P_2 sur leur composante commune A. Les entrées $\underline{IIc}$ correspondent à une transition de P_2 sur sa composante propre C pendant que P_1 ne fait pas de transition. L'entrée $\underline{Ibc}$ correspond à une transition simultanée de P_1 sur sa composante propre B et de P_2 sur sa composante propre C. On voit que la composition parallèle obtenue correspond au produit des systèmes de transitions, synchronisés sur leurs actions communes [1].

5 Un exemple de décomposition parallèle

Dans cette section, nous étudions au moyen d'un exemple le problème de la décomposition parallèle. Ce problème consiste à résoudre pour X des équations de la forme $Q \parallel X = P$. L'exemple que nous traitons est tiré de [2]. Dans cet article, c'est aussi le problème de la résolution de $A \parallel X = B$ qui est abordé. Toutefois, le type de décomposition visé est différent.

Supposons deux chambres contiguës pourvues des portes A, B et C (voir la partie gauche du diagramme ci-dessous). Des personnes peuvent entrer et sortir par ces portes. La notation $A_\triangleright$ ($A_\triangleleft$) indique un passage par la porte A de la gauche vers la droite (de la droite vers la gauche), et similairement pour les portes B et C. Le système de transitions décrit les mouvements permis. Les sommets du graphe indiquent le nombre de personnes présentes dans chaque chambre; ce nombre ne doit pas excéder 1. On remarque aussi qu'une seule porte peut être utilisée à la fois, car il n'y a pas de transition entre les sommets 00 et 11 (cette transition est possible seulement par l'exécution simultanée de $A_\triangleright$ et $C_\triangleleft$). La matrice correspondant au système de transitions est donnée à côté de celui-ci.



Appelons P la relation qui décrit le système de transitions. Nous considérons que l'ensemble des états a trois composantes, soit A, B et C. Les actions $A_{\triangleleft}$ et $A_{\triangleright}$ sont des relations sur l'ensemble A, auxquelles on impose les contraintes

5.1 $A_{\triangleleft} \neq \emptyset, \quad A_{\triangleright} \neq \emptyset, \quad A_{\triangleleft} \cap A_{\triangleright} = A_{\triangleleft} \cap I = A_{\triangleright} \cap I = \emptyset;$

les composantes B et C sont traitées de la même manière. Il serait aussi possible d'utiliser six composantes, une pour chaque action. La relation P est donc une relation sur $A \times B \times C \times V$, avec $V = \{00, 01, 10, 11\}$. La matrice de P est la suivante :

$$\textbf{5.2} \quad P = \begin{pmatrix} \emptyset & \underline{IIC_{\triangleleft}} & \underline{A_{\triangleright}II} & \emptyset \\ \underline{IIC_{\triangleright}} & \emptyset & \underline{IB_{\triangleleft}I} & \underline{A_{\triangleright}II} \\ \underline{A_{\triangleleft}II} & \underline{IB_{\triangleright}I} & \emptyset & \underline{IIC_{\triangleleft}} \\ \emptyset & \underline{A_{\triangleleft}II} & \underline{IIC_{\triangleright}} & \emptyset \end{pmatrix}_{ABC} .$$

Nous voulons décomposer la relation P comme une composition parallèle de relations telles que les processus correspondant à ces relations contrôlent chacun une chambre (*i.e.*, les entrées et les sorties par les portes de cette chambre). Notons que ce problème se généralise à n chambres, mais que la méthode de résolution est la même.

Soit donc la relation P_1 suivante :

$$P_1 = \begin{pmatrix} \emptyset & \underline{A_{\triangleright}I} \cup \underline{IB_{\triangleleft}} \\ \underline{A_{\triangleleft}I} \cup \underline{IB_{\triangleright}} & \emptyset \end{pmatrix}_{AB} .$$

C'est une relation sur l'ensemble des états $A \times B$. L'élément (1,2) de la matrice permet des entrées par les portes A ou B, alors que l'élément (2,1) permet les sorties. Nous voulons résoudre pour P_2 l'équation $P = P_1 \parallel P_2$, c'est-à-dire, par 3.4,

5.3 $P = \pi_1(P_1 \cup I)\hat{\pi}_1 \cap \pi_2 P_2 \hat{\pi}_2 \cup \pi_1 P_1 \hat{\pi}_1 \cap \pi_2 \hat{\pi}_2.$

Il nous faut trouver les relations π_1, π_2 et P_2 qui satisfont l'équation 5.3. On s'aperçoit rapidement que l'ensemble des états de P_2 doit être $A \times B \times C$. En effet, si c'était $B \times C$, la définition de $P_1 \parallel P_2$ (3.4) permettrait des transitions simultanées sur les composantes (portes) A et C, ce qui n'est pas permis par P. Il faut donc que P_2 soit une relation sur $A \times B \times C \times V_2$, où V_2 contient deux points de contrôle (puisque P a quatre points de contrôle et que P_1 en a deux). Par conséquent, nous avons, à une permutation près (voir les commentaires qui suivent 4.3) :

$$\textbf{5.4} \quad \pi_1 = \begin{pmatrix} \omega_1 & \emptyset \\ \omega_1 & \emptyset \\ \emptyset & \omega_1 \\ \emptyset & \omega_1 \end{pmatrix}, \qquad \pi_2 = \begin{pmatrix} I & \emptyset \\ \emptyset & I \\ I & \emptyset \\ \emptyset & I \end{pmatrix},$$

où $\omega_1 : A \times B \times C \to A \times B$. En utilisant ces projections, les lois 3.2 et les contraintes 5.1, on obtient

$$P_1 \cap \hat{\pi}_1 \pi_2 \hat{\pi}_2 \pi_1 = \begin{pmatrix} \emptyset & \underline{A_{\triangleright}I} \cup \underline{IB_{\triangleleft}} \\ \underline{A_{\triangleleft}I} \cup \underline{IB_{\triangleright}} & \emptyset \end{pmatrix}_{AB} \cap \begin{pmatrix} \underline{II} & \underline{II} \\ \underline{II} & \underline{II} \end{pmatrix}_{AB} = \emptyset$$

(remarque : ceci ne dépend pas de la permutation choisie pour π_1 et π_2). De ce résultat, on déduit $\widehat{\pi}_1 P \pi_1 \subseteq P_1 \cup I$:

$$
\begin{array}{ll}
 & \widehat{\pi}_1 P \pi_1 \\
= & \qquad \{ \text{ Équation 5.3. } \} \\
 & \widehat{\pi}_1(\pi_1(P_1 \cup I)\widehat{\pi}_1 \cap \pi_2 P_2 \widehat{\pi}_2 \cup \pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 \widehat{\pi}_2)\pi_1 \\
= & \qquad \{ \text{ Lois 2.2(8,9). } \} \\
 & \widehat{\pi}_1(\pi_1(P_1 \cup I)\widehat{\pi}_1 \cap \pi_2 P_2 \widehat{\pi}_2)\pi_1 \cup \widehat{\pi}_1(\pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 \widehat{\pi}_2)\pi_1 \\
= & \qquad \{ \text{ Définitions 2.3 et 2.5, loi 2.4(2). } \} \\
 & (P_1 \cup I) \cap \widehat{\pi}_1 \pi_2 P_2 \widehat{\pi}_2 \pi_1 \cup P_1 \cap \widehat{\pi}_1 \pi_2 \widehat{\pi}_2 \pi_1 \\
= & \qquad \{ \text{ Résultat ci-dessus. } \} \\
 & (P_1 \cup I) \cap \widehat{\pi}_1 \pi_2 P_2 \widehat{\pi}_2 \pi_1 \\
\subseteq & P_1 \cup I.
\end{array}
$$

Exprimons π_1 comme la composition des projections Ω_1 et ρ_1, *i.e.* $\pi_1 = \Omega_1 \rho_1$ (voir 4.4), où

$$
\begin{array}{l}
\Omega_1 : A \times B \times C \times V_1 \times V_2 \to A \times B \times V_1 \times V_2 \quad \text{et} \\
\rho_1 : A \times B \times V_1 \times V_2 \to A \times B \times V_1.
\end{array}
$$

En combinant cette factorisation de π_1 à la condition précédente, c'est-à-dire $\widehat{\pi}_1 P \pi_1 \subseteq P_1 \cup I$, on a

$$
\begin{array}{ll}
 & \widehat{\pi}_1 P \pi_1 \\
= & \qquad \{ \ \pi_1 = \Omega_1 \rho_1 \text{ et 2.2(15). } \} \\
 & \widehat{\rho}_1 \widehat{\Omega}_1 P \Omega_1 \rho_1 \\
= & \qquad \{ \text{ Définition de } P \text{ (5.2. } \} \\
 & \widehat{\rho}_1 \widehat{\Omega}_1 \begin{pmatrix} \emptyset & \underline{IIC_\lhd} & \underline{A_\rhd II} & \emptyset \\ \underline{IIC_\rhd} & \emptyset & \underline{IB_\lhd I} & \underline{A_\rhd II} \\ \underline{A_\lhd II} & \underline{IB_\rhd I} & \emptyset & \underline{IIC_\lhd} \\ \emptyset & \underline{A_\lhd II} & \underline{IIC_\rhd} & \emptyset \end{pmatrix}_{ABC} \Omega_1 \rho_1 \\
= & \qquad \{ \text{ Définition de } \Omega_1 \text{ (4.4), 3.2, 4.1 et remarque 4.2. } \} \\
 & \widehat{\rho}_1 \begin{pmatrix} \emptyset & \underline{II} & \underline{A_\rhd I} & \emptyset \\ \underline{II} & \emptyset & \underline{IB_\lhd} & \underline{A_\rhd I} \\ \underline{A_\lhd I} & \underline{IB_\rhd} & \emptyset & \underline{II} \\ \emptyset & \underline{A_\lhd I} & \underline{II} & \emptyset \end{pmatrix}_{ABC} \rho_1 \\
\subseteq & \qquad \{ \text{ Valeur de } P_1 \cup I. \ \} \\
 & \begin{pmatrix} \underline{II} & \underline{A_\rhd I} \cup \underline{IB_\lhd} \\ \underline{A_\lhd I} \cup \underline{IB_\rhd} & \underline{II} \end{pmatrix}_{AB}.
\end{array}
$$

Pour satisfaire la dernière inclusion, il n'y a qu'un choix possible pour ρ_1 et ρ_2; ce choix détermine π_1 et π_2 :

5.5 $\rho_1 = \begin{pmatrix} I & \emptyset \\ I & \emptyset \\ \emptyset & I \\ \emptyset & I \end{pmatrix}, \qquad \pi_1 = \Omega_1 \rho_1 = \begin{pmatrix} \omega_1 & \emptyset \\ \omega_1 & \emptyset \\ \emptyset & \omega_1 \\ \emptyset & \omega_1 \end{pmatrix}, \qquad \rho_2 = \pi_2 = \begin{pmatrix} I & \emptyset \\ \emptyset & I \\ I & \emptyset \\ \emptyset & I \end{pmatrix}.$

Nous voulons résoudre pour P_2 l'équation 5.3 (nous connaissons maintenant P, P_1, π_1 et π_2). Dans un premier temps, cherchons $\pi_2 P_2 \widehat{\pi}_2$. L'équation 5.3 a la forme $A = B \cap X \cup C$, avec $A := P$, $B := \pi_1(P_1 \cup I)\widehat{\pi}_1$, $X := \pi_2 P_2 \widehat{\pi}_2$ et $C := \pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 \widehat{\pi}_2$. Mais

$$
\begin{aligned}
& A = B \cap X \cup C \\
\Leftrightarrow\ & A \subseteq B \cap X \cup C \ \wedge\ B \cap X \cup C \subseteq A \\
\Leftrightarrow\ & \qquad \{ \text{ 2.2(4) et autres lois booléennes. } \} \\
& A \cap \overline{C} \subseteq B \cap X \ \wedge\ B \cap X \subseteq A \ \wedge\ C \subseteq A \\
\Leftrightarrow\ & \qquad \{ \text{ 2.2(4) et autres lois booléennes. } \} \\
& A \cap \overline{C} \subseteq B \ \wedge\ A \cap \overline{C} \subseteq X \ \wedge\ X \subseteq A \cup \overline{B} \ \wedge\ C \subseteq A \\
\Leftrightarrow\ & A \cap \overline{C} \subseteq B \ \wedge\ C \subseteq A \ \wedge\ A \cap \overline{C} \subseteq X \subseteq A \cup \overline{B}.
\end{aligned}
$$

Ainsi donc, il y a une solution X pourvu que les conditions suivantes soient satisfaites :

1. $A \cap \overline{C} \subseteq B$: Par 3.2, 4.1 et la remarque 4.2, on a

$$
B = \pi_1(P_1 \cup I)\widehat{\pi}_1 = \begin{pmatrix} \omega_1 & \emptyset \\ \omega_1 & \emptyset \\ \emptyset & \omega_1 \\ \emptyset & \omega_1 \end{pmatrix} \begin{pmatrix} \underline{II} & \underline{A_\triangleright I} \cup \underline{IB_\triangleleft} \\ \underline{A_\triangleleft I} \cup \underline{IB_\triangleright} & \underline{II} \end{pmatrix}_{AB} \begin{pmatrix} \widehat{\omega}_1 & \widehat{\omega}_1 & \emptyset & \emptyset \\ \emptyset & \emptyset & \widehat{\omega}_1 & \widehat{\omega}_1 \end{pmatrix}
$$

$$
= \begin{pmatrix} \underline{IIL} & \underline{IIL} & \underline{A_\triangleright IL} \cup \underline{IB_\triangleleft L} & \underline{A_\triangleright IL} \cup \underline{IB_\triangleleft L} \\ \underline{IIL} & \underline{IIL} & \underline{A_\triangleright IL} \cup \underline{IB_\triangleleft L} & \underline{A_\triangleright IL} \cup \underline{IB_\triangleleft L} \\ \underline{A_\triangleleft IL} \cup \underline{IB_\triangleright L} & \underline{A_\triangleleft IL} \cup \underline{IB_\triangleright L} & \underline{IIL} & \underline{IIL} \\ \underline{A_\triangleleft IL} \cup \underline{IB_\triangleright L} & \underline{A_\triangleleft IL} \cup \underline{IB_\triangleright L} & \underline{IIL} & \underline{IIL} \end{pmatrix}_{ABC}.
$$

   En utilisant l'expression de P (5.2), il est facile de vérifier que $A \cap \overline{C} \subseteq A = P \subseteq B$.

2. $C \subseteq A$:

$$
\begin{aligned}
& C = \pi_1 P_1 \widehat{\pi}_1 \cap \pi_2 \widehat{\pi}_2 \\
= & \begin{pmatrix} \emptyset & \emptyset & \underline{A_\triangleright IL} \cup \underline{IB_\triangleleft L} & \underline{A_\triangleright IL} \cup \underline{IB_\triangleleft L} \\ \emptyset & \emptyset & \underline{A_\triangleright IL} \cup \underline{IB_\triangleleft L} & \underline{A_\triangleright IL} \cup \underline{IB_\triangleleft L} \\ \underline{A_\triangleleft IL} \cup \underline{IB_\triangleright L} & \underline{A_\triangleleft IL} \cup \underline{IB_\triangleright L} & \emptyset & \emptyset \\ \underline{A_\triangleleft IL} \cup \underline{IB_\triangleright L} & \underline{A_\triangleleft IL} \cup \underline{IB_\triangleright L} & \emptyset & \emptyset \end{pmatrix} \cap \\
& \begin{pmatrix} \underline{III} & \emptyset & \underline{III} & \emptyset \\ \emptyset & \underline{III} & \emptyset & \underline{III} \\ \underline{III} & \emptyset & \underline{III} & \emptyset \\ \emptyset & \underline{III} & \emptyset & \underline{III} \end{pmatrix} \\
= & \emptyset.
\end{aligned}
$$

La condition $C \subseteq A$ est donc satisfaite.

3. $A \cap \overline{C} \subseteq A \cup \overline{B}$: Cette condition est trivialement vraie.

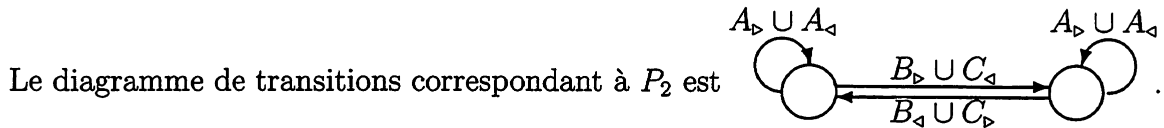
La relation X que nous cherchons existe donc. Elle satisfait la condition $A \cap \overline{C} \subseteq X \subseteq A \cup \overline{B}$. Nous nous contenterons ici de choisir la solution minimale, soit $X = A \cap \overline{C}$; par l'item 2 ci-dessus, $C = \emptyset$, d'où $X = A$, c'est-à-dire $\pi_2 P_2 \widehat{\pi}_2 = P$. Or, par les propriétés des projections (2.5),

$$\pi_2 P_2 \widehat{\pi}_2 = P \;\Rightarrow\; \widehat{\pi}_2 \pi_2 P_2 \widehat{\pi}_2 \pi_2 = \widehat{\pi}_2 P \pi_2 \;\Leftrightarrow\; P_2 = \widehat{\pi}_2 P \pi_2.$$

La relation P_2 que nous cherchons est donc $\widehat{\pi}_2 P \pi_2$: Par 3.2, 5.2 et 5.5,

$$P_2 = \widehat{\pi}_2 P \pi_2 = \begin{pmatrix} I & \emptyset & I & \emptyset \\ \emptyset & I & \emptyset & I \end{pmatrix} \begin{pmatrix} \emptyset & \underline{IIC_\triangleleft} & \underline{A_\triangleright II} & \emptyset \\ \underline{IIC_\triangleright} & \emptyset & \underline{IB_\triangleleft I} & \underline{A_\triangleright II} \\ \underline{A_\triangleleft II} & \underline{IB_\triangleright I} & \emptyset & \underline{IIC_\triangleleft} \\ \emptyset & \underline{A_\triangleleft II} & \underline{IIC_\triangleright} & \emptyset \end{pmatrix}_{ABC} \begin{pmatrix} I & \emptyset \\ \emptyset & I \\ I & \emptyset \\ \emptyset & I \end{pmatrix}$$
$$= \begin{pmatrix} \underline{A_\triangleleft II} \cup \underline{A_\triangleright II} & \underline{IB_\triangleright I} \cup \underline{IIC_\triangleleft} \\ \underline{IB_\triangleleft I} \cup \underline{IIC_\triangleright} & \underline{A_\triangleleft II} \cup \underline{A_\triangleright II} \end{pmatrix}_{ABC}.$$

Le diagramme de transitions correspondant à P_2 est



.

Nous avons décomposé P comme le produit parallèle $P_1 \parallel P_2$, où P_1 utilise les composantes (portes) A et B. Nous aimerions maintenant factoriser la relation P_2 obtenue, de manière à introduire une nouvelle relation qui utiliserait seulement les composantes B et C. Choisissons la relation P_{21}, semblable à P_1, mais sur les composantes B et C :

$$P_{21} = \begin{pmatrix} \emptyset & \underline{B_\triangleright I} \cup \underline{IC_\triangleleft} \\ \underline{B_\triangleleft I} \cup \underline{IC_\triangleright} & \emptyset \end{pmatrix}_{BC}.$$

Nous cherchons la relation P_{22} telle que $P_2 = P_{21} \parallel P_{22}$. En appliquant la même méthode que ci-dessus, on trouve :

$$P_{22} = (\underline{A_\triangleleft II} \cup \underline{A_\triangleright II} \cup \underline{IB_\triangleleft I} \cup \underline{IB_\triangleright I} \cup \underline{IIC_\triangleleft} \cup \underline{IIC_\triangleright})_{ABC}.$$

Cette relation utilise toutes les composantes (A, B, C). Si on la voit comme un processus, son rôle est de s'assurer qu'une seule porte est utilisée à la fois. En effet, un terme comme $\underline{A_\triangleleft II}$, par exemple, signifie qu'un passage vers la gauche par la porte A est fait, alors que les portes B et C ne sont pas utilisées.

Nous avons donc obtenu la décomposition $P = P_1 \parallel P_{21} \parallel P_{22}$, où P_1 contrôle les portes A et B, P_{21} contrôle les portes B et C, et où P_{22} assure qu'une seule porte est utilisée à la fois.

6 Conclusion

L'exemple de résolution de $Q \parallel X = P$ que nous avons traité est très simple. À partir de cette base, nous envisageons d'examiner des cas plus complexes :

- Solution de $Q \parallel X \approx P$, où $\approx$ est une équivalence sur les relations qui dénotent des programmes (par exemple, la bisimulation). Ceci permettrait d'aborder des problèmes comme celui de la conversion de protocoles [3] et de voir si l'approche relationnelle pourrait apporter une contribution intéressante.
- Étude de programmes avec des variables. Ceci est certainement possible, étant donné que la notion de diagramme de programme (3.1) a déjà été utilisée pour traiter des programmes séquentiels avec des variables [10, 11]. L'étude du raffinement de spécifications en programmes parallèles avec des variables devrait permettre de faire des liens intéressants avec nos travaux antérieurs sur le raffinement des spécifications [5, 6, 8, 9].
- Dépendant de la situation à modéliser (rendez-vous, mémoire partagée, communication par messages, etc.), il peut être nécessaire de définir des produits parallèles différents de celui qui est introduit dans cet article. Nous avons commencé à définir de tels produits et à les comparer.

Remerciements

Cette recherche est supportée par Les Recherches Bell-Northern Ltée, le FCAR (Québec) et le CRSNG (Canada), dans le cadre du programme *Action concertée sur les méthodes mathématiques pour la synthèse de systèmes informatiques.*

Bibliographie

[1] A. Arnold. *Systèmes de transitions finis et sémantique de processus communicants.* Masson, Paris, 1992.

[2] A. Bergeron. A unified approach to control problems in discrete event processes. *RAIRO Informatique Théorique et Applications 27*, 6, 1993.

[3] K.L. Calvert et S.S. Lam. Formal methods for protocol conversion. *IEEE J. on Selected Areas in Communications 8*, 1, janvier 1990, pp. 127–142.

[4] L.H. Chin et A. Tarski. Distributive and modular laws in the arithmetic of relation algebras. *University of California Publications 1*, 1951, 341–384.

[5] J. Desharnais, A. Jaoua, F. Mili, N. Boudriga et A. Mili. A relational division operator : The conjugate kernel. *Theoret. Comput. Sci. 114*, 1993, 247–272.

[6] J. Desharnais, A. Mili et F. Mili. On the mathematics of sequential decompositions. *Sci. Comput. Program. 20*, 1993, 253–289.

[7] R.D. Maddux. On the derivation of identities involving projection functions. Department of Mathematics, Iowa State University, Ames, Iowa, 1993.

[8] A. Mili, J. Desharnais et F. Mili. Relational heuristics for the design of deterministic programs. *Acta Inform. 24*, 3, 1987, 239–276.

[9] F. Mili et A. Mili. Heuristics for the construction of while loops. *Sci. Comput. Program. 18*, 1992, 67–106.

[10] G. Schmidt. Programs as partial graphs I : Flow equivalence and correctness. *Theoret. Comput. Sci. 15*, 1981, 1–25.

[11] G. Schmidt et T. Ströhlein. *Relations and graphs*, EATCS Monographs in Computer Science, Springer-Verlag, Berlin, 1993.

[12] A. Tarski. On the calculus of relations. *J. Symb. Log. 6*, 3, 1941, 73–89.

Proof Theory for μCRL: A Language for Processes with Data*

Jan Friso Groote
Department of Philosophy, Utrecht University
Utrecht, The Netherlands

Alban Ponse
Dep. of Mathematics and Computer Science, University of Amsterdam
Amsterdam, The Netherlands

Abstract

A simple specification language, called μCRL (*micro Common Representation Language*), is introduced. It consists of process algebra extended with abstract data types. The language μCRL is designed such that it contains only basic constructs with a straightforward semantics. It has been developed under the assumption that an extensive and mathematically precise study of these constructs and their interaction will yield fundamental insights that are are essential to an analytical approach of well-known and much richer specification languages. To this end, a simple property language is defined in which basic properties of processes, data and the process/data relationship can be expressed in a formal way. Next a proof system is defined for this property language, comprising a rule for induction, the Recursive Specification Principle, and process algebra axioms. The proof theory thus obtained is designed such that automatic proof checking is feasible. It is illustrated with a case study of a counter.

1 Introduction

To guarantee the reliability of software for distributed systems, the need is felt to have more and better means for structural analysis than the contemporary ones. Simulation is effective on large specifications of systems, but it only serves to show that a system is erroneous and it is not very well suited to show a system correct. Most advanced analysis techniques, for instance those that depend on complete explorations of state spaces, are generally only applicable to very small distributed systems. The reason for this is that contemporary state space explorations seldom use the structure present in specifications and therefore the state spaces tend to become very large (see e.g. [16, 21]). Here we are interested in studying the structure that results from the separation between data and processes.

To this purpose we have defined the language μCRL [12]. This is a simple language that comprises a straightforward mechanism to specify abstract data types using plain equations.

*The work reported herein was supported by the European Communities under RACE project no. 1046, Specification and Programming Environment for Communication Software (SPECS).

Furthermore, it consists of elementary process constructors, taken from process algebra [1]. The language is so simple that its semantics is clear. Yet the language is sufficiently expressive to describe large classes of distributed systems. Therefore μCRL is well-suited for further investigations into the process/data relationship.

In order to express and formally prove properties about distributed systems, a proof theory has been developed for μCRL [13]. This proof theory, introduced and illustrated in the present paper, is intended to enable very precise proofs of the correctness of distributed systems and programs. It is well-known that even the slightest error in a program may have serious consequences. Human-made proofs of the correctness of programs are not necessarily correct. If the proofs are sufficiently precise, they can be checked using an appropriate computer program. This increases the reliability of proofs considerably. For μCRL, proof checking by computer has been done and is described in [20].

The language μCRL and its proof theory have been used in several case studies, and have led to a new stream of research. In the last section an overview is given of the current state of the art. The conclusion that we draw from the current developments around μCRL is that it pays off to restrict oneself to a simple core language. Using μCRL we have gained insight in the process data relationship and we have proved distributed systems correct that were outside the scope of process-algebraic techniques a few years ago.

Acknowledgements. We thank Jan Springintveld for his comments regarding this paper. Furthermore, we thank all those, of which we only mention Jan Bergstra, who have contributed to μCRL and its proof theory via comments and discussion.

2 The specification language μCRL

In μCRL data are specified by equational specifications: one can declare sorts and functions working upon these sorts, and describe the meaning of these functions by equational axioms. Processes are described in the style of CCS [18], CSP [14] or ACP [1], where the particular process syntax has been taken from ACP. Starting point is a set of uninterpreted actions that may be parameterised by data. These actions can represent all kinds of real world activities, depending on the usage of the language. Here we do not introduce the syntax of μCRL in detail, but sketch it by an example (that is used to illustrate the proof system). Consider the μCRL specification E in Table 1. It introduces some μCRL keywords, generally printed in boldface. The sort **Bool**eans with constants (**func**) t and f are obligatory in any μCRL specification. A second data type *Nat* with some functions is specified, where most functions are defined in the **rewrite** section, using the variables declared in **var**. The atomic actions in E are *inc*(rease)and *dec*(rease). Finally two process specifications occur (after **proc**). First the process $Counter(t)$ for any closed data term t over *Nat* is defined. The behaviour of eg. $Counter(0)$ is defined by the process term $inc \cdot Counter(S(0)) + \delta \lhd is\text{-}zero(0) \rhd dec \cdot Counter(P(0))$. Here the $\cdot$ represents *sequential composition* and the $+$ represents *choice*. The ternary operator $. \lhd . \rhd .$ is called the *conditional*; it must be read as then-if-else. Precedence of the operators is $\cdot, . \lhd . \rhd ., +$. In the case of $Counter(0)$ the positive consequence of the conditional is the constant δ, *inaction* or *deadlock*, which by definition is a process term. As δ is redundant in a $+$ context, the process $Counter(0)$ can only perform an

sort	**Bool**
func	t, f :→ **Bool**
sort	*Nat*
func	$0 :\to Nat$
	$S, P : Nat \to Nat$
	$add : Nat \times Nat \to Nat$
	$is\text{-}zero : Nat \to$ **Bool**
var	$x, y : Nat$
rew	$P(0) = 0$
	$P(S(x)) = x$
	$add(x, 0) = x$
	$add(x, S(y)) = S(add(x, y))$
	$is\text{-}zero(0) = \mathsf{t}$
	$is\text{-}zero(S(x)) = \mathsf{f}$
act	*inc dec*
proc	$Counter(x{:}Nat) = inc \cdot Counter(S(x)) +$
	$\delta \lhd is\text{-}zero(x) \rhd dec \cdot Counter(P(x))$
	$X = inc \cdot (X \parallel dec)$

Table 1: An example of a μCRL specification

inc action and evolve into $Counter(S(0))$. In turn, $Counter(S(0))$ can perform an *inc* action and further behave as $Counter(S(S(0)))$, or a *dec* action and evolve into $Counter(P(S(0)))$ (which behaves just as $Counter(0)$ by the second **rewrite** rule of E). It is not hard to see that $Counter(0)$ indeed models a "counter". Then the process X is defined. It is not parameterised, and behaves as follows: first *inc* is executed, and then behaviour is conform $X \parallel dec$, the *parallel composition* of X and *dec*, which in turn represents an interleaved execution of both components.

The proof theory to be introduced now gives the means to *formally prove* (respecting the standard semantics) that the process $Counter(0)$ is *equal to* (bisimilar with) the process $Counter(0) \parallel Counter(0)$, in fact stating that two counters in parallel behave as one counter. Furthermore, it can also be proved that $Counter(0) = X$. Both identities illustrate that proving properties of parallel processes may be reducible to proving properties of process descriptions in which no parallel operators occur, and in which control is partly encoded by simple data parameterisation. This is an important feature of the μCRL proof theory.

There are a few other constructs in μCRL that have not been used in the example. Actions may, just as processes, be parameterised. The *label* of an action is simply its name without its possible parameter instantiation. For instance an action $send(0)$ has *send* as its label. There is a *sum operator*, written as $\sum_{d:D}(p)$ which says that the process p may be executed for each possible value for the variable d of data type D. It is generally used to describe input actions of a process. There are a *left merge operator*, written as $\lfloor\!\lfloor$, and a *communication merge operator*, written as $\mid$, that are auxiliary and which are used for calculation with parallel processes. The process $p \lfloor\!\lfloor q$ expresses that p and q run in parallel, but the first step must come from p. The process $p \mid q$ expresses that p and q run in parallel, but the first actions of p and q must communicate (explained below). Furthermore, there is a *hiding operator*, τ_I, an *encapsulation operator*, ∂_H, and a *renaming operator*, ρ_R. The process $\rho_R(p)$ behaves as p, except that the labels of all actions are renamed according to the function R. The process $\tau_I(p)$ behaves as p, except that all actions in p of which the labels occur also in the set I are renamed to the specially dedicated internal action τ. The process $\partial_H(p)$ behaves as p, except that all actions of which the labels occur in the set H are blocked. There is one other unmentioned feature, which is the way communication is declared. If actions a and b must communicate into c, then this is expressed by specifying

$$\textbf{comm} \quad a \mid b = c.$$

This also specifies that possible parameterised occurrences of a and b communicate if they have equal instantiations. For instance $a(0) \mid b(0) = c(0)$.

For a complete specification of the language μCRL see [13], where the language, its static semantics, its operational semantics and some useful subsets of the language are completely formally described.

3 A property language for μCRL

A language is introduced in which simple properties of specifications can be formally expressed. These properties consist of identities between data or process terms, linked together with propositional connectives.

In order to express properties that a specification may have we use *(data) variables* over the sorts declared, and *(process) variables.* We further introduce *substitutions* to extract the precise instances we are interested in. Because properties always refer to a particular specification and because we are dealing with names in a very precise and restrictive way, we define both these concepts relative to the signature of a specification.

Definition 3.1 (*Data and process variables*). Let E be a specification. A finite set V_d containing elements of the form $\langle d{:}D\rangle$ with d some name is called a *set of data variables over* E iff the name D is declared as a sort in E, d is not a constant, or an unparameterised action or process from E, and for each sort name $D' \not\equiv D$ of E it holds that $\langle d{:}D'\rangle \notin V_d$. If we are not interested in the sort of d, we just say that d is 'a variable from V_d'.

Given a set V_d of data variables over E, a finite set V_p of names is called a set of *process variables over* E *and* V_d iff non of its elements occur as a variable in V_d.

We generally use triples E, V_d, V_p, meaning that E is a (well-formed) specification, V_d is a set of data variables over E, and V_p is a set of process variables over E and V_d. Given E, V_d, V_p, we define many sorted terms that may contain variables. We distinguish two kinds of such terms: data terms and process terms.

Definition 3.2 (*Data terms and process terms*). A *data term over* E, V_d, V_p is either a constant from E, a variable from V_d, or an application of a function from E to data terms over E, V_d, V_p of the appropriate sort. A data term is called *closed* iff it does not contain any variables from V_d. Note that for data terms the actual contents of V_p is not relevant.

A *process term over* E, V_d, V_p is defined inductively:

- $p \circ q$ with $\circ \in \{+, \cdot, \|, \lfloor\!\lfloor, \mid, \triangleleft t \triangleright\}$ and t a data term over E, V_d, V_p of sort **Bool**, is a *process term over* E, V_d, V_p if both p and q are,
- $\sum_{d:D}(p)$ is a *process term over* E, V_d, V_p if p is a process term over

$$E, (V_d \setminus \{\langle d{:}n\rangle \mid n \text{ a sort name}\}) \cup \{\langle d{:}D\rangle\}, V_p \setminus \{d\},$$

- $C_{\{n_1,\ldots,n_m\}}(p)$ with $C \in \{\partial, \tau\}$ is a *process term over* E, V_d, V_p if p is, and the n_i are labels of actions from E,
- $\rho_{\{n_1 \to n'_1,\ldots,n_m \to n'_m\}}(p)$ is a *process term over* E, V_d, V_p if p is, and the n_i are labels of actions from E such that if n_i is an action from E then so is n'_i, and if $n_i{:}S_1 \times \ldots \times S_k$ is an action declaration in E then so is $n'_i{:}S_1 \times \ldots \times S_k$,
- δ and τ are *process terms over* E, V_d, V_p,
- n is a *process term over* E, V_d, V_p if n is an action or a process from E or if $n \in V_p$,
- $n(t_1, \ldots, t_m)$ is a *process term over* E, V_d, V_p if either E contains an action declaration of the form $n{:}S_1 \times \ldots \times S_m$ or a process declaration of the form

$$n(x_1{:}S_1, \ldots, x_m{:}S_m) = q$$

  and any t_i is a data term over E, V_d, V_p of sort S_i.

A process term is *closed* iff it does not contain any variables from V_d or V_p.

Definition 3.3. A *property over* E, V_d, V_p is defined inductively in the following way:

- $\mathcal{F}$ ("Falsum") is a *property over* E, V_d, V_p,
- $t = u$ is a *property over* E, V_d, V_p iff
  - either t and u are data terms over E, V_d, V_p that are of the same sort,
  - or t and u are process terms over E, V_d, V_p,
- $\neg(\phi)$ is a *property over* E, V_d, V_p iff ϕ is a property over E, V_d, V_p,
- $(\phi \circ \psi)$ with $\circ \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ is a *property over* E, V_d, V_p iff both ϕ and ψ are.

In properties we omit brackets according to the convention that $=$ binds stronger than any of the logical operators $\neg, \vee, \wedge, \rightarrow$, $\leftrightarrow$, that $\neg$ binds stronger than any of the logical binary operators, and that $\vee, \wedge$ bind stronger than $\rightarrow, \leftrightarrow$.

Now a *property formula*, the basic means to express that some specification has some property, simply consist of a property that has as an attribute the originating specification and variable sets:

$$\phi \textbf{ from } E, V_d, V_p$$

with ϕ a property over E, V_d, V_p. A property formula ϕ **from** E, V_d, V_p is called *closed* iff ϕ contains neither variables from V_d, nor process variables from V_p.

We have introduced more logical symbols than strictly necessary for expressing the properties we are interested in. We regard the symbols $\rightarrow$ and $\mathcal{F}$ as basic, and use the other symbols as abbreviations in the usual way (cf. [7], and recall that $\neg\phi$ abbreviates $\phi \rightarrow \mathcal{F}$).

4 Logical deductions

A proof system over this property language is defined. This system is given in a natural deduction format because this is close to intuitive reasoning. It contains so called 'logical' axioms and rules, suitable to derive the fundamental properties induced by the equality relation $=$ and by the propositional connectives. Our set-up is based on [22].

A deduction can be seen as a tree of which each node is labelled with a property formula (and possibly the name of a rule which has been applied to obtain the property formula). The leaves of the tree are the *assumptions* (also called hypotheses) of the deduction. We use symbols $\mathcal{D}$, possibly subscripted, for arbitrary deductions. We write

$$\begin{array}{c}\mathcal{D}\\ \psi \textbf{ from } E, V_d, V_p\end{array}$$

to indicate that $\mathcal{D}$ has *conclusion* ψ **from** E, V_d, V_p (so the occurrence ψ **from** E, V_d, V_p is part of $\mathcal{D}$ itself). We use the notation $[\phi$ **from** $E, V_d, V_p]$ for a possibly empty set of occurrences of a property formula ϕ **from** E, V_d, V_p in a deduction, thus

$$\begin{array}{c}[\phi \textbf{ from } E, V_d, V_p]\\ \mathcal{D}\end{array}$$

is a deduction $\mathcal{D}$ with a set $[\phi$ **from** $E, V_d, V_p]$ of assumptions in $\mathcal{D}$. As a rule we assume that $[\phi$ **from** $E, V_d, V_p]$ refers to *all* assumptions of the form ϕ **from** E, V_d, V_p in $\mathcal{D}$.

We define logical deductions inductively.

Definition 4.1. The single-node tree with as label a property formula ϕ **from** E, V_d, V_p is a deduction from the open assumption ϕ **from** E, V_d, V_p. There are no cancelled assumptions.

Let $\mathcal{D}_1, \mathcal{D}_2$ be deductions. A new deduction can be constructed according to the rules in Table 2.

These rules are subject to the following restrictions:

1. In applications of the introduction rule →I and the rule RAA (Reductio Ad Absurdum) *all* open assumptions of the form indicated by [...] are cancelled.
2. In applications of →I, RAA, the reflexivity rule REFL, the variable rule VAR and the substitution rule SUB, the conclusion should be a property formula.
3. In applications of SUB the variable x may not be free in any (uncancelled) hypothesis of $\mathcal{D}_1$.
4. Each application of VAR is restricted to one of the following two cases:
   (a) $V_d \subseteq V_d'$ or $V_d' \subseteq V_d$, and $V_p = V_p'$,
   (b) $V_p \subseteq V_p'$ or $V_p' \subseteq V_p$, and $V_d = V_d'$.

The notation $u[t/x]$ means that t is substituted for x in u. The definition of this notion is fully standard, but care must be taken that the types of t and x are the same, and no name clashes occur with data variables bound by the Σ operator. A full definition is given in [13]. The reflexivity rule REFL has an empty premiss, and is therefore called an 'axiom'. The rule VAR is a structural rule that allows (restricted) replacement of variable sets. In the next section we introduce axioms that specify the minimal variable sets involved. With VAR we can obtain variable sets that are suitable for further derivations.

In most deductions the form of the property formulas itself already determines which rule is being applied. Therefore we often omit the names of the rules in deductions. A method that helps to grasp the structure of a given deduction is to number the occurrences of assumptions which are being cancelled, and to repeat the number near the node where the cancellation takes place. Assumptions which are cancelled simultaneously may be given the same number. However, the numbering of discharged assumptions is redundant: by definition any assumption is cancelled at the earliest opportunity.

Example 4.2. Let ϕ **from** E, V_d, V_p and ψ **from** E, V_d, V_p be two property formulas. We derive

$$\cfrac{\cfrac{\phi \textbf{ from } E, V_d, V_p \ ^{(1)}}{\psi \rightarrow \phi \textbf{ from } E, V_d, V_p}\ \rightarrow\text{I}}{\phi \rightarrow (\psi \rightarrow \phi) \textbf{ from } E, V_d, V_p}\ \rightarrow\text{I, [1]}$$

Here the [1] in '→I, [1]' indicates that the assumption "ϕ **from** E, V_d, V_p $^{(1)}$" is cancelled.

$$\dfrac{\begin{array}{c}[\phi \textbf{ from } E, V_d, V_p] \\ \mathcal{D}_1 \\ \psi \textbf{ from } E, V_d, V_p\end{array}}{\phi \rightarrow \psi \textbf{ from } E, V_d, V_p} \rightarrow\text{I}$$

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \phi \textbf{ from } E, V_d, V_p & \phi \rightarrow \psi \textbf{ from } E, V_d, V_p\end{array}}{\psi \textbf{ from } E, V_d, V_p} \rightarrow\text{E}$$

$$\dfrac{\begin{array}{c}[\neg\phi \textbf{ from } E, V_d, V_p] \\ \mathcal{D}_1 \\ \mathcal{F} \textbf{ from } E, V_d, V_p\end{array}}{\phi \textbf{ from } E, V_d, V_p} \text{RAA} \qquad \dfrac{}{t = t \textbf{ from } E, V_d, V_p} \text{REFL}$$

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \phi[t/x] \textbf{ from } E, V_d, V_p & t = u \textbf{ from } E, V_d, V_p\end{array}}{\phi[u/x] \textbf{ from } E, V_d, V_p} \text{REPL}$$

$$\dfrac{\begin{array}{c}\mathcal{D}_1 \\ \phi \textbf{ from } E, V_d, V_p\end{array}}{\phi[t/x] \textbf{ from } E, V_d, V_p} \text{SUB} \qquad \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \phi \textbf{ from } E, V_d, V_p\end{array}}{\phi \textbf{ from } E, V_d', V_p'} \text{VAR}$$

Table 2: Rules for logical deductions

Definition 4.3 (*Derivability*). Let Γ be a set of property formulas. We write

$$\Gamma \vdash \phi \textbf{ from } E, V_d, V_p$$

iff there is a deduction with all uncancelled assumptions in Γ that has conclusion

$$\phi \textbf{ from } E, V_d, V_p$$

. In this case we say that there is a *proof* of ϕ **from** E, V_d, V_p from Γ. If $\Gamma = \emptyset$ the conclusion is called *logically valid.*

We adopt the following two conventions. If in a derivation only property formulas over fixed E, V_d, V_p are considered, we often leave out the additions '**from** E, V_d, V_p'. Furthermore, a proved result may be further used as a single step in future deductions.

The following lemma provides some standard results (among which the congruence properties of $=$).

Lemma 4.4. *Let E, V_d, V_p be given. It holds that*

1. $\vdash \phi \to (\psi \to \phi)$,
2. $\{t = u\} \vdash u = t$,
3. $\{v = t,\ t = u\} \vdash v = u$,
4. $\{t = u\} \vdash w[t/z] = w[u/z]$,
5. $\{\phi \to \psi,\ \psi \to \chi\} \vdash \phi \to \chi$,
6. $\{\phi \to \psi\} \vdash \neg\psi \to \neg\phi$,
7. $\{\phi \to \psi,\ \neg\phi \to \psi\} \vdash \psi$

where in 4 it is assumed that w is a (process or data) term over E, V_d, V_p and $[t/z]$, $[u/z]$ are substitutions over E, V_d, V_p.

For readability we further introduce the notations

$$\bigvee_{i \in I} \phi_i \textbf{ from } E, V_d, V_p \quad \text{and} \quad \bigwedge_{i \in I} \phi_i \textbf{ from } E, V_d, V_p$$

for iterated finite disjunctions and conjunctions, respectively. We adopt the usual convention that $\bigvee_{i \in \emptyset} \phi_i$ **from** $E, V_d, V_p \stackrel{def}{=} \mathcal{F}$ **from** E, V_d, V_p.

5 Modules for μCRL

As μCRL is based on ACP [1] we follow its methodology and consider 'building blocks' of axioms and rules that describe a feature of concurrency in a certain semantical setting. We call such building blocks *modules.* If $M_1, \ldots, M_n$ are modules, then the notation

$$M_1 + \cdots + M_n + \Gamma \vdash \phi \textbf{ from } E, V_d, V_p$$

expresses that with the axioms and rules from $M_1, \ldots, M_n$ we can derive ϕ **from** E, V_d, V_p with all uncancelled assumptions in some set Γ of property formulas.

The module BOOL. Concerning the standard sort **Bool** we define two axioms, corresponding with the demand that any model of a specification E is *boolean preserving* ([12]):

$$\frac{}{\neg(\mathsf{t}=\mathsf{f}) \textbf{ from } E, \emptyset, \emptyset} \text{ Bool1}$$

$$\frac{}{\neg(b=\mathsf{t}) \rightarrow b=\mathsf{f} \textbf{ from } E, \{\langle b{:}\mathbf{Bool}\rangle\}, \emptyset} \text{ Bool2.}$$

The axiom Bool1 states that the Booleans t and f are considered different in our proof system, and the axiom Bool2 expresses that there are at most two Boolean values, represented by t and f. The two axioms Bool1 and Bool2 form the module BOOL. The following lemma states that the reverse implication in Bool2 is derivable.

Lemma 5.1. *For any specification E it holds that*

$$\text{BOOL} \vdash b=\mathsf{f} \rightarrow \neg(b=\mathsf{t}) \textbf{ from } E, \{\langle b{:}\mathbf{Bool}\rangle\}, \emptyset.$$

The module FACT. The basic identities on data terms are those declared in a specification E. Assume $t=u$ occurs as an axiom in E, i.e. $t=u$ is preceded by the keyword **rew**. Then we have an axiom

$$\frac{}{t=u \textbf{ from } E, V_d, \emptyset} \text{ FACT}$$

where V_d is the set of data variables occurring in t and u. Note that the module consisting of all the FACTs from E is implicitly present in the E occurring in property formulas. Therefore we generally do not mention FACT before the turnstyle, although it may have been used.

The module IND(C). In μCRL it is required that any model for the data part is minimal. In the proof theory this can be captured via induction. Therefore we introduce an induction rule. The induction rule that is given here, is a simplified variant of the induction rule provided in [13]. Both rules are not very satisfactory. In order to nicely formulate an induction rule, we need higher order variables (ranging over property formulas). And in order to use the induction principle for a wide range of purposes, such as simultaneous induction, property formulas should have quantifiers that bind variables.

We start with a preparatory definition.

Definition 5.2 (*Constructors*). Let E be a specification, S the name of a sort occurring in E, and C a subset of the function declarations occurring in E. We say that C is a *constructor set of the sort S* iff all functions in C have target sort S, and any closed data term of sort S can be proved equal to a data term that is obtained from applications of the functions in C and terms not of sort S only.

In general it is not possible to prove that a given set is a constructor set within our framework. Reasons for this are that we can neither express 'existential' properties of data terms, nor that a term is obtained from application of a constructor function. Therefore such a proof must be given on a meta-level (structural induction on closed terms).

Assume that for given E, V_d, V_p we have that $\{\langle x{:}S\rangle\} \subseteq V_d$.

$$C \stackrel{def}{=} \{f_j{:}S_1^j \times \ldots \times S_{l_j}^j \to S \mid 1 \leq j \leq k \;,\; k > 0 \;,\; l_j \geq 0\}$$

be a constructor set of the sort S of cardinality k. We introduce the following induction rule IND(C) that is parameterised by the constructor set C. The induction takes place on the variable x.

$$\frac{\begin{array}{c}\mathcal{D}_j \\ (\bigwedge_{\sigma \in I_j} \sigma(\phi)) \to \phi[f_j(z_1^j, \ldots, z_{l_j}^j)/x] \textbf{ from } E, V_d \cup \{\langle z_n^j{:}S_n^j\rangle \mid 1 \leq n \leq l_j\}, V_p\end{array}}{\phi \textbf{ from } E, V_d, V_p} \; 1 \leq j \leq k$$

where for each $1 \leq j \leq k$ the index set I_j is a set of substitutions over $E, V_d \cup \{\langle z_n^j{:}S_n^j\rangle \mid 1 \leq n \leq l_j\}, V_p$ satisfying:

$$\sigma \in I_j \iff \sigma \text{ is the identity, except that it maps } x \text{ to some element from } \{z_n^j \mid 1 \leq n \leq l_j\}.$$

Note that it is required that in IND(C) all the variables $x, z_1^j, \ldots, z_{l_j}^j$ are pairwise different for all appropriate j.

The module REC. This module is the first of a series that can be used to derive identities between process terms using the originating specification and standard process algebra axioms and rules.

Let for some given E it be the case that $n = p$ is a process declaration in E (i.e. the last keyword preceding $n = p$ is **proc**). Then we have an axiom

$$\frac{}{n = p \textbf{ from } E, \emptyset, \emptyset} \; \text{REC}$$

If $n(x_1{:}S_1, \ldots, x_k{:}S_k) = p$ is a process declaration in E, then we have an axiom

$$\frac{}{n(x_1, \ldots, x_k) = p \textbf{ from } E, \{\langle x_1{:}S_1\rangle, \ldots, \langle x_k{:}S_k\rangle\}, \emptyset} \; \text{REC}$$

Like in the case of FACT we adopt the convention not to denote the module REC before the turnstyle.

The modules pCRL **and** μCRL. Let E be a specification. For any equation ϕ displayed in Table 3,4 and 5 we have an axiom

$$\frac{}{\phi \textbf{ from } E, V_d, V_p} \; \textit{name of } \phi$$

where V_d and V_p are the sets of variables occurring in ϕ. In these tables x, y are process variables and p, q are process terms in which the variable d may occur. The letters $t_1, \ldots, t_n$

and $u_1, \ldots, u_m$ stand for data terms, and $\bar{t}$ for a sequence of data terms. c, c' represent δ, τ or range over (declared) actions $a(t_1, \ldots, t_n)$, where $a(t_1, \ldots, t_n)$ represents a if $n = 0$. $\tilde{\gamma}$ is the pre-communication function such that $\tilde{\gamma}(a, b) = c$ if a rule **comm** $a \,|\, b = c$ appears in the specification E. Otherwise $\tilde{\gamma}(a, b) = \delta$. γ is the symmetrical closure of γ.

A1	$x + y = y + x$	SUM1	$\sum_{d:D}(x) = x$
A2	$x + (y + z) = (x + y) + z$	SUM2	$\sum_{d:D}(p) = \sum_{e:D}(p[e/d])$ provided e not free in p
A3	$x + x = x$		
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$	SUM3	$\sum_{d:D}(p) = \sum_{d:D}(p) + p$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	SUM4	$\sum_{d:D}(p + q) = \sum_{d:D}(p) + \sum_{d:D}(q)$
A6	$x + \delta = x$	SUM5	$\sum_{d:D}(p \cdot x) = \sum_{d:D}(p) \cdot x$
A7	$\delta \cdot x = \delta$	SUM11	$(\forall d \in D\ p = q) \rightarrow \sum_{d:D}(p) = \sum_{d:D}(q)$

Table 3: Axioms for pCRL

SUM6	$\sum_{d:D}(p \mathbin{\lfloor\!\lfloor} z) = \sum_{d:D}(p) \mathbin{\lfloor\!\lfloor} z$	CF	$a(t_1, \ldots, t_n) \,|\, b(u_1, \ldots, u_m) = \begin{cases} \gamma(a,b)(t_1, \ldots, t_n) & \text{if } n = m,\ t_i = u_i,\ \gamma(a,b) \text{ defined} \\ \delta & \text{otherwise} \end{cases}$
SUM7	$\sum_{d:D}(p \,|\, z) = \sum_{d:D}(p) \,|\, z$		
SUM8	$\sum_{d:D}(\partial_H(p(d))) = \partial_H(\sum_{d:D}(p))$		
SUM9	$\sum_{d:D}(\tau_I(p(d))) = \tau_I(\sum_{d:D}(p))$		
SUM10	$\sum_{d:D}(\rho_R(p(d))) = \rho_R(\sum_{d:D}(p))$		
		CD1	$\delta \,|\, x = \delta$
		CD2	$x \,|\, \delta = \delta$
CM1	$x \parallel y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x + x \,|\, y$	CT1	$\tau \,|\, x = \delta$
CM2	$c \mathbin{\lfloor\!\lfloor} x = c \cdot x$	CT2	$x \,|\, \tau = \delta$
CM3	$c \cdot x \mathbin{\lfloor\!\lfloor} y = c \cdot (x \parallel y)$		
CM4	$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$	DD	$\partial_H(\delta) = \delta$
CM5	$c \cdot x \,|\, c' = (c \,|\, c') \cdot x$	DT	$\partial_H(\tau) = \tau$
CM6	$c \,|\, c' \cdot x = (c \,|\, c') \cdot x$	D1	$\partial_H(a(\bar{t})) = a(\bar{t})$ if $a \notin H$
CM7	$c \cdot x \,|\, c' \cdot y = (c \,|\, c') \cdot (x \parallel y)$	D2	$\partial_H(a(\bar{t})) = \delta$ if $a \in H$
CM8	$(x + y) \,|\, z = x \,|\, z + y \,|\, z$	D3	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$
CM9	$x \,|\, (y + z) = x \,|\, y + x \,|\, z$	D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$

Table 4: Primary μCRL-axioms

Interesting are the SUM axioms that appeared first in [13]. SUM1 says that we may omit the Σ operator in $\sum_{d:D}(x)$ as d does not occur in variable x. SUM2 expresses a form of α-conversion. This axiom is superfluous, because it is derivable by definition of substitution

TID	$\tau_I(\delta) = \delta$		RD	$\rho_R(\delta) = \delta$
TIT	$\tau_I(\tau) = \tau$		RT	$\rho_R(\tau) = \tau$
TI1	$\tau_I(a(\bar{t})) = a(\bar{t})$	if $a \notin I$	R1	$\rho_R(a(\bar{t})) = R(a)(\bar{t})$
TI2	$\tau_I(a(\bar{t})) = \tau$	if $a \in I$		
TI3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$		R3	$\rho_R(x + y) = \rho_R(x) + \rho_R(y)$
TI4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$		R4	$\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y)$

Table 5: More μCRL-axioms

[13]. SUM3 says that for each term $\sum_{d:D}(p)$ we may extract a copy of p where the variable d may occur unbounded in p. Although no completeness result for these axioms does exist, they appeared fully adequate for all calculations that we have encountered up till now.

The axioms in Table 3, together with Bool1, Bool2, C1 and C2 are called the pCRL (pico CRL, [4]) axioms. The axioms of pCRL together with the axioms in Table 4 and 5 are called the μCRL axioms.

The module COND. We define for any specification E two axioms characterising the behaviour of the conditional [15]:

$$\frac{}{x \lhd \mathsf{t} \rhd y = x \textbf{ from } E, \emptyset, \{x, y\}}\ \text{C1} \qquad \frac{}{x \lhd \mathsf{f} \rhd y = y \textbf{ from } E, \emptyset, \{x, y\}}\ \text{C2}.$$

These two axioms form the module COND. The following lemma describes two basic properties that can be proved using the module COND. Both these results will be used later in the paper.

Lemma 5.3. *Let E, V_d, V_p be such that $\langle b{:}\mathbf{Bool}\rangle \in V_d$ and $\{x, y, z\} \subseteq V_p$. Then*

1. $\mu\text{CRL} \vdash x + x \lhd b \rhd \delta = x$ **from** E, V_d, V_p,
2. $\text{BOOL} + \text{COND} \vdash (b = \mathsf{t} \to x = y) \to (x \lhd b \rhd z = y \lhd b \rhd z)$ **from** E, V_d, V_p.

The module RSP. In order to derive identities between infinite processes we introduce (an extended version of) the Recursive Specification Principle (RSP, see eg. [1]).

The idea of RSP is that if two (different) process terms both satisfy some 'process-equation', then those process terms are considered equal. In general we use a system of such equations, each of which must contain at its left-hand side a (possibly parameterised) fresh identifier and at its right-hand side a 'process term' that may contain the new identifiers. These identifiers may be parameterised with data. We introduce a mechanism that defines substitution of parameterised process terms in a system of process-equations. The soundness of RSP depends on the guardedness of the system of process-equations used. In the following we make all these notions precise, and introduce the rule RSP.

Let E, V_d, V_p be given and let $n_1, \ldots, n_m$ be m different fresh identifiers. We call a system G of m equations $G_1, \ldots, G_m$ a system of *process-equations over* E, V_d, V_p if each equation G_i has at its left-hand side an expression of the form n_i or of the form

$$n_i(x_{i1}, \ldots, x_{im_i})$$

where any x_{ij} is a data variable from V_d. Moreover, the right hand side of each equation G_i may only contain free variables x_{ij} and the new, properly typed identifiers n_i or $n_i(t_{i1}, \ldots, t_{im_i})$ for $1 \leq i \leq m$ and $1 \leq j \leq m_i$.

Next we introduce a substitution mechanism for a system $G = G_1, \ldots, G_m$ of process-equations over E, V_d, V_p. Abbreviating the (possible) variables of n_i by $\bar{x}_i$, we define

$$G_i[\lambda \bar{x}_i.p(\bar{x}_i)/n_i]$$

as the equation obtained by substituting $\lambda \bar{x}_i.p(\bar{x}_i)$ for the n_i-occurrences in G_i, and then repeatedly performing β-conversion on the respective arguments of the process name n_i. For any identifier without arguments only the substitution of p is performed.

The rule RSP is restricted to systems of process-equations that are (syntactically) guarded. In [5] a stronger variant of RSP, called CL-RSP, is introduced. This notion uses the notion of convergent linear process operators, which is necessary to handle internal actions conveniently. But for the example in this paper the current version of RSP suffices.

Definition 5.4 (*Guardedness of G*). A term p is a *guard* iff:

- $p \equiv \delta$, or
- $p \equiv n(t_1, \ldots, t_n)$ or $p \equiv n$ and n is an action label, or
- $p \equiv q_1 \circ q_2$ with $\circ \in \{+, \triangleleft t \triangleright\}$ and q_1 and q_2 are guards, or
- $p \equiv q_1 \circ q_2$ with $\circ \in \{\cdot, \|, \lfloor\!\lfloor\}$ and q_1 or q_2 are guards, or
- $p \equiv q_1 \mid q_2$, or
- $p \equiv \sum_{x:S}(q_1)$ and q_1 is a guard, or
- $p \equiv C_S(q_1)$ with $C \in \{\partial, \rho\}$ and S being a set of labels, and q_1 is a guard.

Let G be a system of process-equations and let N be the left-hand side of one of the equations of G. We say that N is *guarded* in r, where r is a subterm of one of the right-hand sides of G, iff

- $r \equiv q_1 \circ q_2$ with $\circ \in \{+, \|, \lfloor\!\lfloor, \mid, \triangleleft t \triangleright\}$, and N is guarded in q_1 and q_2,
- $r \equiv q_1 \cdot q_2$ with N is guarded in q_1, and q_1 is a guard or N is guarded in q_2,
- $r \equiv \Sigma_{x:S} q_1$ and N is guarded in q_1,
- $r \equiv C_S(q_1)$ with $C \in \{\partial, \rho\}$ and S being a set of labels (or in the case of ρ a renaming function), and N is guarded in q_1,

- $r \equiv \delta$ or $r \equiv \tau$,
- $r \equiv n'$ for a *name* n' and $N \not\equiv n'$,
- $r \equiv n'(u_1, \ldots, u_{m'})$ and $N \not\equiv n'(x_{i1}, \ldots, x_{im_i})$.

If N is not guarded in r we say that N appears *unguarded* in r.

The *Identifier Dependency Graph* of G, notation $IDG(G)$, is constructed as follows:

- each fully instantiated left-hand side of the equations of G is a node,
- if $N = r \in G$, M appears unguarded in r, N' is obtained by instantiating N via a closed substitution σ, and M' is obtained by instantiating M via σ (where variables in M bound in r are supposed to be different from those in N), then there is an edge $N' \to M'$.

We call G *guarded* iff $IDG(G)$ is well founded, i.e. does not contain an infinite path.

Given a guarded system $G_1, \ldots, G_m$ of m process-equations over E, V_d, V_p, we define the following rule RSP:

$$\frac{\begin{matrix}\mathcal{D}_{1i} & \mathcal{D}_{2i} \\ G_i[\lambda\bar{x}_j.p_j(\bar{x}_j)/n_j]_{j=1}^m \textbf{ from } E, V_d, V_p & G_i[\lambda\bar{x}_j.q_j(\bar{x}_j)/n_j]_{j=1}^m \textbf{ from } E, V_d, V_p\end{matrix}}{p_k(\bar{x}_k) = q_k(\bar{x}_k) \textbf{ from } E, V_d, V_p \quad (1 \le k \le m)} 1 \le i \le m$$

where

- for $1 \le i \le m$ the $p_i(\bar{x}_i)$ and $q_i(\bar{x}_i)$ are process terms over E, V_d, V_p,
- the notation $[\ldots]_{j=1}^m$ abbreviates the m given, simultaneous substitutions.

6 An example: proving some properties of counters

In this section we first illustrate the μCRL proof theory by proving some properties of the specification given in Table 1. As formal proofs (i.e. deductions) of non-trivial facts are often hard to read, and may take in our case a larger space than available on one page, we will not give these. Instead we only write down the essential steps of a proof, trusting that the suggestion of a formal proof is sufficiently clear. Furthermore we will often represent proofs in a linear style: in a context where E, V_d, V_p are fixed, we write $t = u$ if this identity can be obtained by applications of reflexivity, symmetry or substitutivity (see Lemma 4.4.2+4), or via the rule SUB (so no variables that occur free in an open assumption are instantiated). Moreover, based on the transitivity of $=$, proved in Lemma 4.4.3, we write $t_1 = t_2 = \ldots = t_n$ to represent a proof with conclusion $t_1 = t_n$. For convenience we sometimes write names of axioms or identities above the $=$.

The following results are about the specification in Table 1. Note that it is not hard to check that

$$C \stackrel{def}{=} \{0 \colon \to Nat, S \colon Nat \to Nat\}$$

is a set of constructors of sort Nat. First we prove the following relation between the parallel operator and the addition operator *add* on the data type Nat.

Theorem 6.1. *Two counters in parallel behave as one counter with the contents added. Let* $V_d = \{\langle b{:}Nat\rangle, \langle c{:}Nat\rangle\}$. *Then*

$$\mu\text{CRL} + \text{IND}(C) + \text{RSP} \vdash$$

$$Counter(b) \parallel Counter(c) = Counter(add(b,c)) \ \textbf{from}\ E, V_d, \emptyset.$$

Proof. The main step in the proof is an application of RSP. First we define the guarded system G as follows:

$$\begin{aligned} n(b,c) = {} & inc \cdot n(S(b),c) + inc \cdot n(S(c),b) + \\ & \delta \lhd \textit{is-zero}(b) \rhd dec \cdot n(P(b),c) + \delta \lhd \textit{is-zero}(c) \rhd dec \cdot n(P(c),b). \end{aligned}$$

We prove $G[\lambda b,c.Counter(b) \parallel Counter(c)/n]$ **from** $E, V_d, \emptyset$ and $G[\lambda b,c.Counter(add(b,c))/n]$ **from** $E, V_d, \emptyset$. Then by RSP the theorem follows in a straightforward way.

First we show $G[\lambda b,c.Counter(b) \parallel Counter(c)/n]$. This is a straightforward expansion.

$$\begin{aligned} & Counter(b) \parallel Counter(c) \overset{\text{expansion}}{=} \\ & \quad inc \cdot (Counter(S(b)) \parallel Counter(c)) + \\ & \quad (\delta \lhd \textit{is-zero}(b) \rhd dec \cdot Counter(P(b))) \mathbin{\lfloor\!\lfloor} Counter(c) + \\ & \quad inc \cdot (Counter(S(c)) \parallel Counter(b)) + \\ & \quad (\delta \lhd \textit{is-zero}(c) \rhd dec \cdot Counter(P(c))) \mathbin{\lfloor\!\lfloor} Counter(b) \\ & = inc \cdot (Counter(S(b)) \parallel Counter(c)) + \\ & \quad \delta \lhd \textit{is-zero}(b) \rhd dec \cdot (Counter(P(b)) \parallel Counter(c)) + \\ & \quad inc \cdot (Counter(S(c)) \parallel Counter(b)) + \\ & \quad \delta \lhd \textit{is-zero}(c) \rhd dec \cdot (Counter(P(c)) \parallel Counter(b)). \end{aligned}$$

Now we show $G[\lambda b,c.Counter(add(b,c))/n]$.

$$\begin{aligned} & Counter(add(b,c)) \overset{\text{expansion}}{=} \\ & \quad inc \cdot Counter(S(add(b,c))) + \\ & \quad \delta \lhd \textit{is-zero}(add(b,c) \rhd dec \cdot Counter(P(add(b,c))) \\ & = \\ & \quad inc \cdot Counter(add(S(b),c)) + inc \cdot Counter(add(S(c),b) + \\ & \quad \delta \lhd \textit{is-zero}(b) \rhd dec \cdot Counter(add(P(b),c)) + \\ & \quad \delta \lhd \textit{is-zero}(c) \rhd dec \cdot Counter(add(P(c),b)) \end{aligned}$$

by the property $add(S(b),c) = add(S(c),b) = S(add(b,c))$ (by IND(C)), Lemma 5.3, BOOL, COND and $\textit{is-zero}(c) = \mathsf{f} \to P(add(b,c)) = add(P(c),b)$ (and the symmetric variant). □

Corollary 6.2. *Let* $V_d = \{\langle a{:}Nat\rangle, \langle b{:}Nat\rangle, \langle c{:}Nat\rangle\}$. *Then*

$$\mu\text{CRL} + \text{IND}(C) + \text{RSP} \vdash$$

$$(Counter(a) \parallel Counter(b)) = (Counter(b) \parallel Counter(a)) \text{ and}$$

$$\begin{aligned} & (Counter(a) \parallel Counter(b)) \parallel Counter(c) = \\ & \quad Counter(a) \parallel (Counter(b) \parallel Counter(c)) \ \textbf{from}\ E, V_d, \emptyset. \end{aligned}$$

Proof. By commutativity and associativity of the addition operator *add* (standard) and Theorem 6.1. □

We conclude with the following theorem, stating that the process specified by $Counter(0)$ satisfies a standard definition (see [1]).

Theorem 6.3. *The process* $Counter(0)$ **from** E *satisfies*

$$\mu\text{CRL} + \text{IND}(C) + \text{RSP} \vdash Counter(0) = X \textbf{ from } E, \emptyset, \emptyset.$$

Proof. With some intermediate results and RSP we can prove

$$Counter(S(b)) = Counter(b) \parallel dec.$$

Hence $inc \cdot Counter(S(0)) = inc \cdot (Counter(0) \parallel dec)$ by the congruence properties of '=' and instantiation. Using the definitions we obtain

$$Counter(0) = X.$$

□

7 Conclusion

The major advantage of a formal language and a formal proof system is also its major drawback, because one fixes beforehand all allowed means for expressing oneself and for providing proofs. When designing a proof system it is very hard to foresee how it will be used and whether it is appropriate for that usage. We think that experience is the key to an appropriately designed formal system. Thus we have developed the proof system simultaneously with its application to small examples. From these initial examples we concluded that the property language as defined in this document was adequate. But when applying the proof system to larger examples, as described below, we felt that existential and universal quantification would be convenient in the property language. Moreover, quantification should not be restricted to first order quantification over processes and data terms, but also range over higher order objects such as functions (see for instance the sum elimination lemma in [10] that can be more elegantly expressed using higher order quantification). Another problem that we did not foresee is that the hiding-, encapsulation- and renaming operators rename actions independently of their data parameter. Up till now this is satisfactory when writing down specifications, but it causes problems when providing proofs (see [17]).

Despite the problems sketched above we are rather satisfied with the language μCRL, the property language and its proof system. This is a consequence of the developments sketched below. On all fronts we have made considerable advances during the last two years and there is no sign that this will change in the future. As regards the further developments of the proof system, we have clear guidelines for extending the property language, which we expect to be heavily influenced by the lambda-calculus based languages used in contemporary theorem provers [8]. The major problem will be to keep the proof system concise, and yet sufficiently expressive to handle the next generation of proofs.

- First the developments around μCRL have led to some theoretical results. The expressive power of μCRL has been investigated in [19]. Subsets of μCRL have been identified that can generate primitive recursive and recursive transition systems. In [4] the status of invariants in process algebra has been cleared up. It is concluded that invariants are theoretically superfluous, but practically useful. In [4] derived rules are provided that enable to calculate in process algebra with the explicit use of invariants.

  In [2] a version of process algebra with data has been provided where the variable binding of the Σ operator has been avoided by explicitly using combinators. In [9] a version of μCRL extended with time is given.

- Second, μCRL, and in particular its proof theory have led to a yet rather small class of precise logical correctness proofs of distributed systems. In [17] Milner's scheduler provided in [18] is re-verified. It is concluded that the proof in [18] is indeed very precise, but contains a lot of meta notation and reasoning. Furthermore, a condition, namely that a scheduler should always consist of more than one cycler has been forgotten. This is exactly the kind of mistake that we want to eradicate, so we see this as an assuring result. In [11] a bounded retransmission protocol is verified. In [5, 6] sliding window protocols are verified. The last two verifications are particularly interesting because sliding window protocols were generally considered too hard for process-algebraic techniques. Finally, we mention [10] where a bakery protocol is proven correct. In this document many elementary results about calculations with processes and data have been established, among which the fundamental sum elimination lemma.

  The techniques for process-algebraic verifications are developed further by studying more examples that lead to the identification of systematic approaches for providing proofs. The first result of this kind is the above mentioned work on invariants in process algebra [4]. As stated earlier, this article concludes that invariants are theoretically superfluous, but do allow a separation of concerns, which is badly needed in calculations that require many pages. In [5] a technique using *cones* and *foci* has been identified, which is crucial in [6]. It is expected that, when the complexity and size of distributed systems that are proven correct increase, more of these general methods are identified.

- Third, as a result of the very precise logical nature of μCRL and its proof system, proof checking turns out to be possible. The general method for doing so in the proof system `Coq` [8] is outlined in [20]. The first larger example of a proof that was verified in this way is the alternating bit protocol as proved correct in [1]. This is described in [3]. The above mentioned proofs in [11] and [17] have been computer checked, too.

  From these experiments we draw the simple conclusion that proof checking by computer is feasible, but, due to the rather large number of applications of axioms in a verification, it is still a lot of work. We can also conclude that proof checking is useful. In the case studies mentioned above more or less serious mistakes in the proofs have been detected, and minor improvements of the proven theorems have been made.

  Currently, we see as the most important issue of research that proof checking must be made more efficient. We believe that the best methods for doing so is via (higher order) rewriting and more advanced matching techniques.

References

[1] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[2] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with combinators. Report P9319, Programming Research Group, University of Amsterdam, 1993.

[3] M.A. Bezem and J.F. Groote. A formal verification of the alternating bit protocol in the calculus of constructions. Technical Report Logic Group Preprint Series No. 88, Utrecht University, March 1993.

[4] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. Technical Report Logic Group Preprint Series No. 98, Utrecht University, September 1993.

[5] M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. To appear as technical report, Logic Group Preprint Series, Utrecht University, 1993.

[6] M.A. Bezem and J.F. Groote. A correctness proof of a sliding window protocol in μCRL. To appear as technical report, Logic Group Preprint Series, Utrecht University, 1993.

[7] D. van Dalen. *Logic and Structure.* Springer-Verlag, 1983.

[8] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Version 5.8. Technical report, INRIA – Rocquencourt, May 1993.

[9] W.J. Fokkink. A simple specification language combining processes, time and data. Technical Report CS-R9132, CWI, Amsterdam, 1991.

[10] J.F. Groote and H. Korver. A correctness proof of the bakery protocol in μ-CRL. Technical Report 80, Logic Group Preprint Series, Utrecht University, 1992.

[11] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. To appear as Technical Report, Logic Group Preprint Series, Utrecht University, 1993.

[12] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, CWI, Amsterdam, 1990.

[13] J.F. Groote and A. Ponse. Proof theory for μCRL. Report CS-R9138, CWI, 1991.

[14] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[15] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[16] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[17] H. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. Technical report, CWI, Amsterdam, 1993. To Appear.

[18] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[19] A. Ponse. Computable processes and bisimulation equivalence. Report CS-R9207, CWI, Amsterdam, January 1992.

[20] M.P.A. Sellink. Verifying process algebra proofs in type theory. Technical Report Logic Group Preprint Series No. 87, Utrecht University, March 1993.

[21] R. de Simone and D. Vergamini. Aboard AUTO. Technical Report 111, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1989.

[22] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction (vol I).* North-Holland, 1988.

Implementation of Constrained Transition Systems: A Unified Approach *

Antoine Rauzy
LaBRI, Université Bordeaux I,
F-33405 Talence Cedex, FRANCE

Srećko Brlek
LaCIM, Un. du Québec à Montréal,
Montréal, P.Q. H3C 3P8, CANADA

February 1, 1994

Abstract

In this paper we present an extension of the Arnold-Nivat model for concurrent communicating processes. The extension given here consists in adding constraints of a certain type to the labelled transitions. The main feature of this model is its simplicity, capturing many of the examples found in the literature. Part of the model has been implemented, especially constraints defined over finite domains. The real case is in the process of being implemented.

1 Introduction

Real-time requirements in the specification of telecommunications services has brought the introduction of new models such as timed transition systems [1, 19], hybrid systems [16, 21], constrained automata [14], among others.

All of them deal with the fundamental issue of incorporating time intervals in the design of a system in order to solve the problem of scheduling the tasks in such a manner that the system under analysis behave "correctly". This means that we have sometimes to verify properties such as on one hand, the absence of deadlocks, livelocks, or other undesirable properties, and on the other hand the presence of all desirable functionalities.

These types of problems arise naturally in the domain of telephone services, especially when dealing with "feature interaction".

Our approach will be based on the Arnold-Nivat model [6], which was introduced to take into account he behaviour of communicating concurrent processes. The constrained approach based on Binary Decision Diagrams (BDD) allows us to compute effciently properties about the system, by pruning the unnecessary computations. The tool used for implementing our model is TOUPIE [22].

*Research supported by the grant FCAR-BNR-CRSNG, "Action Concertée sur les Méthodes Mathématiques pour la Synthèse des Systèmes Informatiques"

The paper is organized as follows. We recall the definition of the Arnold-Nivat model in section 2. Then we add constraints over arbirtrary domains in section 3. We shall not discuss here algorithmic and complexity issues that will be presented in a forthcoming paper. Indeed a whole theory of constrained automata is needed as in classical automata theory to provide the closure properties of languages recognized by these automata: union, intersection, projections, morphisms, substitutions, and so on. Decidability of properties have also to be set in this context and efficient algorithms have to be developped. Section 4 is concerned with the implementation of the model in an adequate language based on the μ-calculus, named TOUPIE. In section 5, we apply this model to a typical example, showing the expressive power of TOUPIE.

2 The Arnold-Nivat Model

Let Σ be a finite alphabet of actions. A transition system labelled by Σ is a 5-tuple $\langle S, T, \alpha, \beta, \lambda \rangle$ (denoted sometimes $\langle S, T \rangle$ for short) where:

- S is a set of states, T is a set of transitions;
- $\alpha, \beta : T \to S$ are maps that give respectively, the source and target of each transition;
- $\lambda : T \to \Sigma$ is a map which labels every transition of T by a letter in Σ.

The set of initial states will be denoted by I. A path c is a sequence $(t_1, \ldots, t_n)$ of transitions such that $\forall i \geq 1, \beta(t_i) = \alpha(t_{i+1})$. The maps α and β extend naturally to paths by defining $\beta(c) = \beta(t_n)$, and $\alpha(c) = \alpha(t_1)$.

If $A_i = \langle S_i, T_i, \alpha_i, \beta_i, \lambda_i \rangle, i = 1, \ldots, n$ are transition systems labelled respectively by Σ_i, then $V \subseteq \Sigma_1 \times \cdots \times \Sigma_n$ is called a *synchronization constraint*. The *synchronized product* [6, 3] $\mathcal{A}$ of the A_i, is denoted by

$$\mathcal{A} = \langle A_1, \ldots, A_n; V \rangle,$$

and defined as the transition system $\langle S, T, \alpha, \beta, \lambda \rangle$ where

- $S = S_1 \times \cdots \times S_n$,
- $T = \{\langle t_1, \ldots, t_n \rangle \mid \langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle \in V\}$,
- α, β, λ are respectively the products of the $\alpha_i, \beta_i, \lambda_i$.

This model was implemented [2], and used successfully for instance in the verification and design of simple call processing systems [4] and also in the analysis of deadlocks [5].

3 Adding Constraints to the Model

Let $A = \langle S, T \rangle$ be a transition system and $\{D_j | j = 1..m\}$ a family of sets not necessarily finite, called *domains*. The cartesian product is denoted as usual by $D = \prod_{i=1}^{m} D_i = D_1 \times \ldots \times D_m$, and variables ranging in D will be denoted by $\vec{y}$.

This approach can be found for instance in [19, 14] where a constraint C_t is attached with each transition $t \in T$, with source state $\alpha(t) = s$, interpreted as follows:

(1) $$s \models_{\mathcal{A}} C_t \Longrightarrow t \text{ is enabled}$$

For sake of clarity, we shall omit the subscript $\mathcal{A}$ if no confusion could arise.

The behaviour of the automaton will be computed as follows. Given an initial state $s_0 \in S$, and an initial value $\vec{y} \in D$, the *constrained reachability* of A is obtained recursively by the rules:

(R1) $s \models_{\mathcal{A}} C_t \Longrightarrow t$ is enabled

(R2) $\beta(t)$ is given a value in D

Note that we do not make a precise statement about how R1 and R2 are computed: one could think about C_t as a local arithmetic constraint as in [14], or as a predicate depending on some non-local trace property occurring on the path leading to the current, or on a property occurring in the future. This procedure may contain two sorts of nondeterminism. The first concerns the classical notion of nondeterministic automata, that is, when there exists two (or more) transitions with same source state s, labelled by the same letter but having different target states. It is easy to see that such nondeterminism can be removed by a determinisation algorithm which glues all the target states and taking the disjunction of the constraint conditions. The second type arises when s satisfies two different constraints, enabling consequently two different transitions. In this case, the computation of the behaviours requires to store the current values in s. Indeed, if one computes the behaviours up to a given length, (for instance by a breadth-first traversal algorithm), then the intermediate values have to be stored because a node could eventually be visited many times. Therefore we need to introduce some local variables for each $s \in S$, leading to the following definition.

Definition 1 *A* c*-transition system* $A_c = \langle S', T' \rangle$ *is defined as a transition system for which*

(i) $S' = \{(s, \vec{y}_s) \mid s \in S\}$;

(ii) $T' = \{(t, C_t) \mid t \in T\}$.

The major breakthrough of adding constraints is that one transition can represent many instances of the domains values as we shall see in the next section, provided the fact that we have an adequate language for expressing constraints, and with the benefit of reducing the size of the transition systems.

We caracterize *admissible* transitions by the following condition,

$$t \text{ is } \textit{admissible} \iff \alpha(t) \models_{\mathcal{A}} C_t \ ,$$

and extend the notion to paths by composition of transitions in the same manner as in classical automata theory [17].

The *language recognized* by the c-transition system $A' = \langle S, T' \rangle$ is defined as usual by

$$L(A') = \{\lambda(p) \mid p \in (T')^*, \text{and } p \text{ is admissible}\}$$

Remark that it could happen that some states are no longer reachable from the initial state, and it is often convenient, for practical reasons, to consider only the reachable states of the c-transition system.

Fact 2 $[\forall t,\ \Psi(t) = \prod_{i=1}^m D_i] \implies [A' = A]$.

In this case, the right-hand side of (C) is always true. It is therefore interpreted as if there were no constrained transitions in the system, and we clearly have the classical model.

3.1 The case of finite domains

In a c-transition system $A'_c = \langle S', T' \rangle$ (based on $A = \langle S, T \rangle$), domain variables can take different values depending on the initial value, initial state and traversal, that can be described by indexing the visit of a state with natural numbers. Let N be the set of integers and Φ and Ψ be some partial functions

$$\text{(1)} \qquad \Phi : S \times N \to \prod_{i=1}^{m} D_i \qquad\qquad \Psi : T \times N \to \wp(\prod_{i=1}^{m} D_i)$$

where $\wp$ is the power set function. The function Ψ is often called a *substitution*, and, for all $t \in T$, $\Psi(t)$ is called a *constraint* on t. Therefore a constraint is given by some relation $C \subseteq \prod_{i=1}^m D_i$.

Recall that a transition is enabled if the following condition holds

$$\text{(C)} \qquad t \in T' \iff \Phi(\alpha(t,n)) \in \Psi(t,n).$$

If $|\Psi(t,n)| = 1$ for all $t \in T$ and $n \in N$, then each transition represents one instance of the domain variable. Therefore, an initialization is given by defining $\Phi(s_0, 1)$, and the map Φ is defined from the behaviour of the c-transition system. The graph obtained describes the evolution of the variables and in the case where the domain D is finite, the constrained model is equivalent to the original Arnold-Nivat model.

Indeed, it suffices to add all the necessary states and transitions to take into account all the possible solutions to (C). Define a transition system $V = \langle S_v, T_v \rangle$, where states are given by all the possible values of the domain variable and transitions are given by condition (C). Then, the c-transition system A'_c has the same behaviour as the synchronized product of A and V.

In this case, the constraints capture the following state equivalence of the synchronized product.

$$s_1 \sim s_2 \iff \exists s \in S_v, \exists t_1, t_2 \in T_v$$

such that

(i) $s = \alpha(t_1) = \alpha(t_2)$;

(ii) $\beta(t_i) = s_i$, $i = 1, 2$;

(iii) $\lambda(t_1) = \lambda(t_2)$;

(iv) $\Phi(s_1, n) = \Phi(s_2, n)$.

3.2 The synchronized product.

Let $\{A_{\mathbf{c},i}\} = \{\langle S'_i, T'_i\rangle, i = 1, \ldots, n\}$, be a finite family of $\mathbf{c}$-transition systems. The *free* product of the $\{A_{\mathbf{c},i}\}$ is the $\mathbf{c}$-transition system $P_{\mathbf{c}} = \langle S', T'\rangle$ where

- $S' = S'_1 \times \cdots \times T'_n$,
- $T' = T'_1 \times \cdots \times T'_n$,,
- α, β, λ are respectively the products of the $\alpha_i, \beta_i, \lambda_i$.

The synchronized product of finite $\mathbf{c}$-transition systems $A_1, A_2, \ldots A_n$, with synchronization constraint V, is obtained from the free product above by requiring that a transition t satisfies the following conditions

(i) $t \in V$;

(ii) $\forall i \in [1..n], \vec{y}_{t,i} \in C_{t,i}$.

4 Implementation in a Constraint Language

We have experimented the previously described model within the constraint language Toupie [13]. Toupie implements an extension of the propositional μ-calculus to finite domains. The design of Toupie follows two influences: from Symbolic Model Checking, it takes the computational model (the μ-calculus), and thus, it is close to tools such as SMV [20]; From Constraint Logic Programming [15], it takes the will to be a full language as well as constraint solving techniques. As both it uses Binary Decision Diagrams [8] to represent and manipulate Boolean functions [9, 7], but it extends this technique to finite domains.

Toupie has been already used to perform efficient abstract interpretation of Prolog programs [11] and to verify mutual exclusion algorithms [12].

4.1 Syntax and Semantics of Toupie Programs

Presenting the full syntax and denotational semantics of Toupie would be somewhat boring here, so we limit ourselves to what is necessary to understand the example of the next section.

Variables, Constants, Domains and Atomic Relations : In Toupie, there are *variables* denoted by identifiers beginning with an upper case letter (as in Prolog) and *constants* that are either symbolic and denoted by identifiers beginning by a lower case letter or numeric (i.e. integral). The variables are typed, i.e., belong to *domains* which are (small) sets of constants or (small) ranges. The domain of a variable must be declared with its first occurrence. A type declaration is in the form: `X : { k1, ..., kn}` or `X : i..j` where `X` is a variable the `ki`'s are constant symbols and `i` and `j` are integers (and thus `i..j` denotes the corresponding range).

It is possible to name domains by issuing a command like:

```
let d = domain {a,b,c}
```

and, following this command, the identifier `d` stands in the sequel of the program for the domain `{a,b,c}`.

With (typed) variables and constants, one can build *atomic relations*: `(X1=X2)` or `(X1=k)` or `(X1#X2)` or `(X1#k)` where `X1` and `X2` are variables and `k` is a constant symbol (`#` stands for the nonequality $\neq$).

For variables with a numerical domain, it is possible to build more complicated atomic relations, i.e. linear inequations and systems of linear inequations. For instance:

```
(3*X - 5*Y >= 6*Z + 2)
{X<Y, Y<Z, Z-X=3}
```

Toupie is an interpreter. Here follows a possible session:

```
2p |= let dom = domain 1..5

2p |= (X:dom = Y:dom + 3) ?
{X=4,Y=1}
{X=5,Y=2}
```

"2p |=" is the Toupie prompt. In response to the query `(X:dom = Y:dom + 3) ?`, Toupie enumerates the elements of the cartesian product of the domains of variables satisfying the constraint. An equivalent form for this query is:

```
2p |= lambda (X:dom,Y:dom) (X=Y+3) ?
```

Formulae : With atomic relations one can build *formulae*:

- An atomic relation is a formula.
- The two Boolean constants `0` and `1` are formulae.
- If `f` and `g` are formulae then so are `~f` ($\neg f$), `f & g` ($f \wedge g$), `f | g` ($f \vee g$), `f => g`, ($f \Longrightarrow g$),...
- If `X1,..., Xn` are variables and `f` is a formula, then so are `forall X1,..., Xn f` and `exist X1,..., Xn f`.

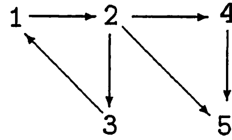


Figure 1: An automaton

- Predicate calls are also formulae (see next paragraph).

Another possible session:

```
2p |= let dom = domain 1..5

2p |= lambda (X:dom,Y:dom)
        exist U:dom ( {X<U,U<Z} & forall V:dom ({X<V,V<Z} => (U=V)) ) ?
{X=1,Y=3}
{X=2,Y=4}
{X=3,Y=5}
```

Predicates : A Toupie program is a set of *predicate definitions* having different identifiers for head or the same identifier but different arities. Predicates are defined as least or greatest fixpoints of equations (for the inclusion in the powerset of the cartesian product of the domains of their formal parameters). Predicate definitions are as follows:

`p(X1,...,Xn) += f` or `P(X1,...,Xn) -= f` where p is an n-ary predicate symbol, `X1`,..., `Xn` are individual variables, and `f` is a formula. The tokens `+=` and `-=` denotes respectively least and greatest fixpoint definition.

Note that the fixpoint definitions must be monotonic in order to ensure the existence of fixpoints. This condition could be easily checked syntactically.

Example : Assume that we want to compute some properties of the automaton depicted in Fig. 1.

The first step consists in defining a binary predicate g that models the automaton. g contains exactly the pairs (α, β) such that there exists an edge between the vertex α and the vertex β :

```
2p |= let state = domain 1..5

2p |= g(X:state,Y:state) += (
          ((X=1) & (Y=2))
        | ((X=2) & (Y in {3,4}))
        | ((X=3) & (Y=1))
        | ((X=4) & (Y=5)) )
```

```
2p |= g(1,Y:state) ?
{Y=2}

2p |= g(X:state,1) ?
{X=3}

2p |= lambda (X:state,Y:state)   exist Z:state (g(X,Z) & g(Z,Y)) ?
{X=1,Y=3}
{X=2,Y=1}
{X=2,Y=5}
{X=3,Y=2}
```

Note that g is defined here as a least fixpoint. However, since there is no recursive call, it could be defined as a greatest fixpoint as well.

Now, one may want to compute, for instance, the states that are reachable from the state 2. This can be done by defining a new predicate as follows:

```
2p |= reachable_from_2(X:state) += (
          edge(2,X)
        | exist Y:state (reachable_from_2(Y) & g(Y,X))
        )
2p |= reachable_from_2(X:state) ?
1
```

The response 1 means that for every value of X the relation is satisfied, and thus that every state is reachable from the state 2.

Records : As we shall see, an individual automaton is in general modeled by defining a ternary relation encoding its transitions, i.e. triples of the form : < source state, transition label, target state >. A synchronized product of k automata is modeled by defining a $3 \times k$-ary relation <source states, transition labels, target states>. In order to avoid to write k variables each time a global state or a global transition label is manipulated, it is possible to declare vectors/records/lists:

```
let state1 = domain 0..1
let state2 = domain 1..3
let global_state = list (S1:state1, S2:state2)
```

Thereafter, it is possible to declare a record variable of type `global_state`.

```
2p |= lambda (^G:global_state) (G.S1=G.S2) ?
{G.S1=1, G.S2=1}

2p ->  1 displayed solution
```

Such a composite variable G is prefixed by ^ when it is declared or manipulated per se. Its "fields" are accessible as shown above : `G.S1`, `G.S2`. For more explanations on records, see [22].

Variable Ordering : From the first paper by R. Bryant [8], it is well known that the size of a decision diagram (binary or not) crucially depends on the indices chosen for the variables: he gives an example where the associated BDD can be either linear or exponential w.r.t. the number of variables depending on the variable indexing.

By default, in Toupie, the variables are indexed with a very simple heuristic, known for its rather good accuracy. It consists in crossing the formula considered as a syntactic tree with a depth-first left-most procedure and to number variables in the induced order.

Nevertheless, this heuristic can produce very poor performances due to the projection operation. This operation is performed each time a predicate `p(X1,...,Xn)` is called since the result of the computation of the corresponding fixpoint must be projected on arguments of the call, here `X1,...,Xn`. This operation can be dramatically inefficient if the arguments are not ordered as the formal parameters.

This is the reason why, the user is allowed to define its own indices by `X@i`, where `X` is the first occurrence of a variable and `i` is any integer. For composite variables, indices declarations are in the form `^X@i!j`, meaning that the first field of `X` must be numbered `i`, the second one `i+j`, the third one `i+2j` and so on.

4.2 Decision Diagrams

The Decision Diagrams used in Toupie to encode relations are an extension for symbolic finite domains of the Bryant's Binary Decision Diagrams (see [8] for an outstanding presentation of BDDs). We just mention here the differences between DDs and BDDs (see [13] for a more detailed discussion).

The main difference between the two representations stems from the fact that a DD node is n-ary (and not binary), where n is the cardinality of the domain to which the variable belongs. Such a node encodes a *case* connective:

Definition 3 *Let X be a variable in $\{k_1, \ldots, k_r\}$, and let $f_1, \ldots, f_r$ be some formulae. Then:*

$$case(X, f_1, \ldots, f_r) = ((X = k_1) \wedge f_1) \vee \ldots \vee ((X = k_r) \wedge f_r)$$

The important properties of BDDs : reduction, negation in constant time, canonicity are preserved by this extension.

Example 4 Let `X` and `Y` be two variables in `{0..2}` . The DD encoding the constraint `(X + Y <=1)` is shown in Fig. 2 (complemented edges are marked with a black dot).

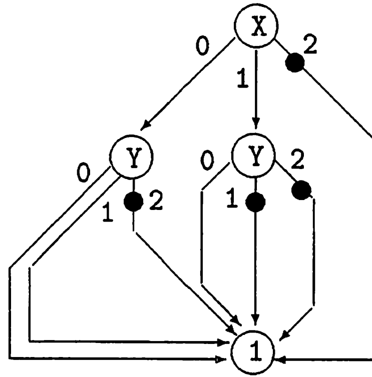


Figure 2: DD associated with $(X : 0..2 + Y : 0..2 <= 1)$

Logical Operations on DDs. The induction principle that permits to perform logical operations between DDs is the same as for BDDs. A unique "If-Then-Else" connective is used :

Definition 5 *Let* $p = case(X, p_k, \ldots, p_k)$, $q = case(X, q_1, \ldots, q_k)$ *and* $r = case(X, r_1, \ldots, r_k)$ *be three DDs. Then,*

$$\begin{aligned} ITE(p,q,r) &= (p \wedge q) \vee (\neg p \wedge r) \\ &= case(X, ITE(p_1, q_1, r_1), \ldots, ITE(p_k, q_k, r_k)) \end{aligned}$$

It is easy to induce an effective procedure from this principle. Note however that quantifications, projections, and constraint solving use different mechanisms (again see [13] for a more detailed discussion).

Compacted representation. When dealing with variables with rather large domains (it could be the case especially for numerical variables) many outedges of a node may lead to the same son. A compacted representation is allowed in Toupie, by using a declaration of the form `X:0..10:compacted`. In this case, outedges of nodes labeled with `X` are labeled with ranges rather than with individual constants (such a DD is shown in Fig. 3).

The "good" properties of DDs are preserved: negation in constant time, canonicity. Logical operations can be even accelerated by this coding.

Note finally that such a representation allows the coding of approximations of relations between variables taking their values in dense domains : there is no need for `X` to be a natural number in the DD pictured Fig. 3. It could be a real variable as well.

5 A Commented Example

Description. In order to illustrate the expressive power of Toupie and its practical efficiency, let us consider the problem of designing a protocol between a resources dispatcher

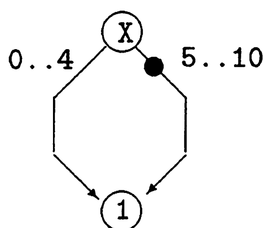


Figure 3: DD associated with $(X : 0..10 < 5)$

and a number of buffers. At the beginning, the dispatcher owns a given number of resources and the buffers are empty. During the process, each buffer tries to get one by one a number of resources from the dispatcher. When it has obtained them, it performs an action (no matter what this action actually is), then starts to give them back to the dispatcher (still one by one). When it is empty, it performs another action and starts to get resources again. And so on infinitely. The protocol must ensures some properties such as deadlock freeness, safety, fairness ...

Let us examine first a very simple version in which the dispatcher non deterministically gives a resource to a buffer or get a resource from a buffer without any additional control. In order to make the Toupie code sufficiently small to be easily readable we consider the case where there are only two buffers.

The idea we use, first proposed by Madre and Coudert [10] and Mac Millan & al [7], is to encode transition systems in a symbolic way.

Individual Processes. The behavior of the dispatcher is thus modeled by means of labeled transition system. The states are the possible numbers of resources the dispatcher has, and the transitions model its actions, i.e. `get` (a resource from a buffer), `put` (a resource in a buffer) and `e` (when it remains idle). This transition system is encoded as an automaton in the same manner as it was done in section 4.1. The variable `S`, `L` and `T` stand respectively for sources, labels and targets of transitions.

```
let resources = 5  /* initial number of resources */
let dispatcher_state = domain 0..resources
let dispatcher_label = domain {e,get,put}

dispatcher(S:dispatcher_state,L:dispatcher_label,T:dispatcher_state) += (
     ((L=e)   & (T=S))
   | ((L=get) & (T=S+1))
   | ((L=put) & (T=S-1))   )
```

The behavior of the two buffers is modeled in the same way. Here, states are pairs (section,current number of resources), where section is a Boolean variable indicating whether the buffer tries to get (`up`) or to put back (`down`) a resource. The actions performed by the buffer when it is full or empty are modeled by means of the same transition label `tau`.

```
let maxsize = 5     /* maximum size of the buffers */
let buffer_size    = domain 0..maxsize
let buffer_section = domain {up,down}
let buffer_label   = domain {e,get,put,tau}
let buffer_state   = list ( Size:buffer_size,
Section:buffer_section      )

buffer(^S@1!1:buffer_state, L@3:buffer_label, ^T@4!1:buffer_state) += (
    ((L=e)   & (T.Size=S.Size)  & (T.Section=S.Section))
  | ((L=get) &
       (S.Section=up) & {S.Size<maxsize, T.Size=S.Size+1} & (T.Section=up))
  | ((L=put) &
       (S.Section=down) & {S.Size>0, T.Size=S.Size-1} & (T.Section=down))
  | ((L=tau) & (
         ((S.Section=up) & {S.Size=maxsize, T.Size=S.Size} & (T.Section=down))
       | ((S.Section=down) & {S.Size=0, T.Size=S.Size} & (T.Section=up))
       )) )
```

Note that variables indices are fixed in the above predicate. `S.Size`, `S.Section`, `L`, `T.Size`, `T.Section` receive respectively the indices 1, 2, 3, 4 and 5. In the remaining, we keep these index declarations in order to provide a faithful Toupie session, but we don't discuss these choices here. A discussion on this subject can be found in [18] and [13].

Synchronized Product. Now, one must synchronize the three processes, that is to constrain, for instance, the dispatcher to give a resource (transition `put`) when the first buffer gets it (transition `get`) while the second one remains idle (transition `e`). These synchronization constraints are written as a Toupie predicate:

```
let label = list ( D :dispatcher_label,
               B1:buffer_label,
               B2:buffer_label )

synchronizator(^L@3!5:label) += (
      ((L.D=e)   & (L.B1=tau) & (L.B2=e))
    | ((L.D=e)   & (L.B1=e)   & (L.B2=tau))
    | ((L.D=get) & (L.B1=put) & (L.B2=e))
    | ((L.D=get) & (L.B1=e)   & (L.B2=put))
    | ((L.D=put) & (L.B1=get) & (L.B2=e))
    | ((L.D=put) & (L.B1=e)   & (L.B2=get)) )
```

The initial state and edges of the synchronized product are obtained as follows:

```
let state = list ( D      :dispatcher_state,
               ^B1@0!1:buffer_state,
               ^B2@0!1:buffer_state )

initial(^S@1!5:state) += ((S.D=resources) & (S.B1.Size=0) & (S.B2.Size=0))
```

```
edge(^S@1!5:state, ^T@4!5:state) +=
       exist ^L@3!5:label (
          dispatcher(S.D,L.D,T.D)
      & buffer(S.^B1, L.B1, T.^B1)
      & buffer(S.^B2, L.B2, T.^B2)
      & synchronizator(^L)  )
```

There are tuples of individual states that do not correspond to reachable states of the synchronized product. The set of reachable states is computed by means of a least fixpoint, starting from the initial state and traversing the automaton:

```
reachable(^T@4!5:state) += (
      initial(^T)
    | exist ^S@1!5: state (reachable(^S) & edge(^S,^T)) )
```

Deadlocks. The predicates `reachable` and `edge` allow the verification of properties of the system. Let us recall that a deadlock (in a weak sense) is a reachable state from which no transition is possible or only a transition leading in a deadlockstate. The Toupie program to detect deadlocks is as follows:

```
deadlock(^S@1!5:state) += (
    reachable(^S)
  & forall ^T@4!5:state (transition(^S,^T) => deadlock(^T)) )

2p |= deadlock(^T@4!5:state) ?
lambda (^S@1!5:state) deadlock(^S)
{S.D=0, S.B1.Size=1, S.B1.Section=up, S.B2.Size=4, S.B2.Section=up}
{S.D=0, S.B1.Size=2, S.B1.Section=up, S.B2.Size=3, S.B2.Section=up}
{S.D=0, S.B1.Size=3, S.B1.Section=up, S.B2.Size=2, S.B2.Section=up}
{S.D=0, S.B1.Size=4, S.B1.Section=up, S.B2.Size=1, S.B2.Section=up}
{S.D=1, S.B1.Size=1, S.B1.Section=up, S.B2.Size=3, S.B2.Section=up}
{S.D=1, S.B1.Size=2, S.B1.Section=up, S.B2.Size=2, S.B2.Section=up}
{S.D=1, S.B1.Size=3, S.B1.Section=up, S.B2.Size=1, S.B2.Section=up}
{S.D=2, S.B1.Size=1, S.B1.Section=up, S.B2.Size=2, S.B2.Section=up}
{S.D=2, S.B1.Size=2, S.B1.Section=up, S.B2.Size=1, S.B2.Section=up}
{S.D=3, S.B1.Size=1, S.B1.Section=up, S.B2.Size=1, S.B2.Section=up}

2p ->  10 displayed solutions
```

There are 10 deadlock states (in this toy example). A quick analysis shows that the problem arises when both buffers try to get (up) resources while the dispatcher has not enough resources to satisfy at least one of them.

The protocol must be modified to avoid this situation. A simple way to do this is to add the constraint that the dispatcher never gives a resource to a buffer if it has not enough resources to satisfy its request. This is done by modifying the synchronization constraints in the following way:

```
synchronizator(^S@1!5:state, ^L@3!5:label) += (
      ((L.D=e)   & (L.B1=tau) & (L.B2=e))
    | ((L.D=e)   & (L.B1=e)   & (L.B2=tau))
    | ((L.D=get) & (L.B1=put) & (L.B2=e))
    | ((L.D=get) & (L.B1=e)   & (L.B2=put))
    | ((L.D=put) & (L.B1=get) & (L.B2=e)   & (S.D>=maxsize-S.B1.Size))
    | ((L.D=put) & (L.B1=e)   & (L.B2=get) & (S.D>=maxsize-S.B2.Size))  )
```

The protocol shows now to be deadlock free.

Fairness. An important property to be verified by the protocol is that a buffer that asks resources will always obtain these resources. Such a fairness property can be expressed as the greatest fixpoint of a least fixpoint. With the least one, we compute the set of states S such that every path leaving S goes to a state in which the first buffer is full (from symmetry arguments it suffices to consider only the first buffer). With the greatest one, we remove from the set the states that are not on infinite loops composed by states of the set.

```
transition(^S@1!5:state,^T@4!5:state) += (reachable(^S) & edge(^S,^T))

live(^S@1!5:state) += (reachable(^S) & ~deadlock(^S))

to_full_buffer1(^S@1!5:state) += (
     live(^S)
   & forall ^T@4!5:state
       (transition(^S,^T) => ((T.B1.Size=maxsize) | to_full_buffer1(^T))) )

fair_state(^S@1!5:state) -= (
     to_full_buffer1(^S)
   & forall ^T@4!5:state (transition(^S,^T) => fair(^T)) )

2p |= fair_state(^S@1!5:state) ?
lambda (^S@1!5:state) fair(^S)
0
```

This shows that the protocol is not fair. Actually, it is possible to make it fair, but it would be too long to present the new protocol here. Anyhow, the fairness property is interesting both from practical and theoretical points of view since it requires a greatest fixpoint computation.

Performances. The tables below indicate the running times on a SPARC 1 IPX workstation with 48 megabytes of memory for various number of buffers, sizes of buffers and initial number of resources (the 3 numbers written in column heads indicate these numbers respectively).

With a non-compacted representation:

	2/10/10	2/10/15	2/10/20	3/10/10	3/10/20	3/10/30	4/10/10	4/10/20
states	84	386	484	328	8328	10648	656	?
reachable	0s86	1s66	2s03	1s55	10s50	12s60	2s40	?
fairness	1s78	2s60	2s21	4s31	9s81	10s11	9s00	?

With a compacted representation:

	2/10/10	2/10/15	2/10/20	3/10/10	3/10/20	3/10/30	4/10/10	4/10/20
states	84	386	484	328	8328	10648	656	?
reachable	0s50	1s25	1s53	0s93	11s30	12s96	1s35	?
fairness	0s78	1s48	1s05	2s11	6s00	4s75	4s00	?

With a non-compacted representation:

	5/5/5	5/5/10	5/5/15	5/5/20	5/5/25
states	832	58944	189696	244768	248832
reachable	0s85	10s63	22s28	30s05	28s26
fairness	3s93	17s23	23s06	28s45	30s05

With a compacted representation:

	5/5/5	5/5/10	5/5/15	5/5/20	5/5/25
states	832	58944	189696	244768	248832
reachable	0s70	10s28	20s01	20s61	20s55
fairness	2s88	12s00	11s01	11s05	11s03

These examples show that Toupie can handle rather large examples. It is not surprising that limitations are due to lack of memory and not to excessive running times: it is in general the case with BDDs. It is also interesting to point out that the compacted representation really improves the performances.

6 Conclusion

Constrained transitions systems appear as a convenient model for describing the behaviour of timed transition systems. This extension of the classical Arnold-Nivat model can be efficiently implemented in Toupie and challenging algorithmic problems arise in the finite case as in the real case with the computation of the constraints and remain to be investigated.

References

[1] R. ALUR AND T. HENZINGER, "Real-time System = Discrete System + Clock Variables." *Proc. AMAST93*, Iowa City, November 1993.

[2] A. ARNOLD, " MEC: a system for constructing and analysing transition systems", In J. Sifakis, editor, *Automatic Verification of Finite State Systems*, Lect. Notes in Comp. Sci., Vol. **407**, pp 117–132, Springer-Verlag, 1989.

[3] A. ARNOLD, "Systèmes de transitions finis et sémantique des processus communicants." *Technique et Science Informatiques*, AFCET, Vol. **9**, No. 3, 1990.

[4] A. ARNOLD AND S. BRLEK "Conception d'un système de gestion d'appels téléphoniques : un exemple d'utilisation de méthodes formelles", *In G.V. Bochmann, Ed., Actes du CFIP93*, Hermès, pp 149–163, 1993.

[5] A. ARNOLD AND S. BRLEK "Automatic Deadlock Analysis in an E-Mail System", submitted to *Software Paractice and Experience*, 1993.

[6] A. ARNOLD AND M. NIVAT, "Comportements de Processus," *Colloque AFCET "Les Mathématiques de l'Informatique"*, pp 35–68, 1982.

[7] J.R. BURCH, K.L. MCMILLAN, D.L. DILL AND L.J. HWANG, "Symbolic Model Checking: 10^{20} States and Beyond", IEEE transactions on computers, 1990.

[8] R. BRYANT, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", ACM Computing Surveys, 1992.

[9] W. BUETTNER AND H. SIMONIS, "Embedding Boolean Expressions into Logic Programming", Journal of Symbolic Computation, Vol. 4, pp 191–205, 1987.

[10] O. COUDERT AND C. BERTHET AND J-C. MADRE, "Verification of Synchronous Sequential Machines Based on Symbolic Execution", In *Automatic Verification Methods for Finite State System,* J. Sifakis, Ed., Lect. Notes in Comp. Sci. vol. **407**, Springer-Verlag, 1989.

[11] M-M. CORSINI, B. LE CHARLIER, K. MUSUMBU AND A. RAUZY, "Efficient Abstract Interpretation of Prolog Programs by means of Constraint Solving over Finite Domains", in Proceedings of the 5th Int. Symposium on Programming Language Implementation and Logic Programming, PLILP'93, 1993.

[12] M-M. CORSINI, A. GRIFFAULT AND A. RAUZY, "Yet another Application for Toupie: Verification of Mutual Exclusion Algorithms", in Proceedings of Logic Programming and Automated Reasonning, LPAR'93, Lect. Notes in Comp. Sci., Springer-Verlag, 1993.

[13] M-M. CORSINI AND A. RAUZY. "Symbolic Model Checking and Constraint Logic Programming: a Cross-Fertilization", *Proceedings of the European Symposium on Programming ESOP'94,* Don Sannella ed., 1994.

[14] L. FRIBOURG AND M.V. PEIXOTO, "Concurrent Constraint Automata." *Research Report LIENS 93-10*, Ecole Normale Sup., Paris, 1993.

[15] P. VAN HENTENRYCK, "Constraint Handling in Logic Programming" Logic Programming Serie, MIT Press, 1990.

[16] T. HENZINGER, X. NICOLLIN, J. SIFAKIS, AND S. YOVINE, "Symbolic Model Checking for Real Time Systems." In *Proc. of the 7th Annual Symposium on Logic in Computer Science*, pp 394–406, IEEE Computer Society Press, 1992.

[17] S. EILENBERG, "Automata, Languages, and Machines", Vol. A. Academic Press, 1974.

[18] R. ENDERS AND T. FILKORN AND D. TAUBNER, "Generating BDDs for Symbolic Model Checking in CCS", Journal of Distributed Computing, Vol. **6**, pp 155–164, Springer Verlag, 1993.

[19] Y-J. LIN AND G. WUU, "A Constrained Approach for Temporal Intervals in the Analysis of Timed Trasitions", *Proc. of the XI-th IFIP*, pp 215–230, Elsevier, North-Holland, 1991.

[20] K. MAC MILLAN, "Symbolic Model Checking" Kluwer Academic Publishers, Boston/ Dordrecht/London, 1993.

[21] X. NICOLLIN, A. OLIVERO, J. SIFAKIS AND S. YOVINE, "An Approach to the Description and Analysis of Hybrid Systems", In A. Ravn and H. Rischel, Eds, *Workshop on Hybrid Systems*, Lect. Notes in Comp. Sci. Springer-Verlag, 1993.

[22] A. RAUZY, "Toupie Version 0.25 : User's Manual", Technical Report, LaBRI - URA CNRS 1304 - Université Bordeaux I, 1994.

Deux nouvelles techniques d'utilisation de MEC

Alain Griffault
LaBRI* Université Bordeaux I

1 Introduction

Considérons les types de données suivants :
Une tâche est un composant logiciel caractérisé par les paramètres suivants:

- une priorité définie par une valeur entière;
- une périodicité définie par une valeur réelle T;
- une durée d'exécution définie par une valeur réelle d.

Une CPU est un composant matériel caractérisé par son traitement des interruptions, à savoir:

- les interruptions sont interdites;
- les interruptions sont toujours possibles;
- les interruptions sont fonctions des priorités des tâches;

L'étude décrite dans cet article a été réalisée dans le but de répondre au problème suivant: Etant donné un ensemble de tâches partageant une unique CPU; est-il possible de trouver avec le vérifieur MEC [Arn89], un ensemble de contraintes sur les paramètres des tâches et de la CPU afin que le comportement global soit sans blocage.

Ce problème est issu directement d'une application industrielle dans laquelle la chronologie du développement est la suivante :

1. Choix de la CPU;
2. Ecriture des tâches : *la durée d'exécution est fixée*;
3. Choix des priorités des tâches;
4. Calcul des périodicités.

*Unité de Recherche Associée au Centre National de la Recherche Scientifique n° 1304

Afin de respecter cette approche, nous avons procédé en deux étapes : d'une part une étude sur les trois premiers points au cours de laquelle les blocages détectés sont considérés comme des blocages de contrôle; d'autre part une étude du dernier point au cours de laquelle les blocages détectés sont considérés comme des blocages temporels.
Chacune de ces étapes a apporté ses problèmes, et nous a permis ainsi de mettre en place deux nouvelles méthodes d'utilisation de MEC :

1. La multiplicité des hypothèses (une hypothèse correspond à une configuration des paramètres) rend la comparaison (non pas en terme d'équivalence, mais d'existence de comportements communs) des différents comportements globaux malaisée. Nous avons résolu ce problème en considérant toutes les hypothèses comme des restrictions par rapport à une hypothèse de référence. Pour l'outil MEC, cette idée se traduit par le calcul d'un seul produit synchronisé, sur lequel on applique des filtres de sélection des états et des transitions.

2. Les périodicités des tâches pouvant être très différentes les unes par rapport aux autres (de la micro-seconde à la journée), une modélisation de ces périodes par la technique habituelle des compteurs aurait abouti à une explosion du nombre d'états du graphe d'accessibilité. Afin de résoudre ce problème, nous proposons une méthode en deux points:

    - Remplacer les contraintes temporelles sur les périodicités par des contraintes symboliques plus faibles;
    - Exprimer ces contraintes symboliques, non pas par des systèmes de transitions que l'on synchronise avec le comportement global, mais par des fonctions points fixes que l'on calcule sur le comportement global.

L'article est un exemple de mise en oeuvre de ces méthodes, sur ce problème simplifié de tâches qui partagent une CPU.

2 Le contexte du problème

2.1 Description informelle du problème

Soit N tâches numérotées de 1 à N, activées par des horloges, qui partagent une CPU.
Supposons également que les périodicités T des différentes tâches vérifient l'ordonnancement $T1 < T2 < T3 < T4.... < TN$
Le problème consiste à trouver des contraintes suffisantes pour qu'il n'y ait ni blocage de contrôle, ni blocage temporel. Les hypothèses et contraintes que nous avons étudiées sont les suivantes:

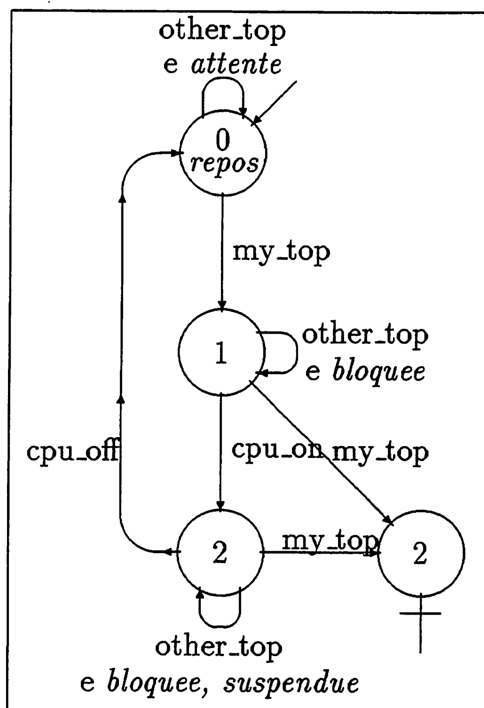
- L'ordre des priorités des tâches est il important ?
- Les droits d'interruptions de la CPU sont ils importants ?
- Les relations entre les périodicités des tâches et les durées d'exécution de ces tâches sont elles importantes ?

2.2 Modélisation par systèmes de transitions et synchronisations

2.2.1 Une tâche

Une tâche est activée par un top d'horloge (my_top), elle acquiert alors dès que possible la CPU (cpu_on), pour la rendre dès son travail effectué (cpu_off). Sur chaque état elle perçoit et réagit à tous les tops d'horloge émis (other_top).

Les marques de transitions *attente, bloquee, suspendue* permettent de différencier et d'interdire par calcul les trois non actions d'une tâche (e). Elles seront utilisées pour restreindre les calculs en fonction des hypothèses.
La marque d'état *repos* indique les états dans lesquels le temps peut s'écouler librement. Elle sera utilisée ultérieurement comme paramètre pour les fonctions points fixes.

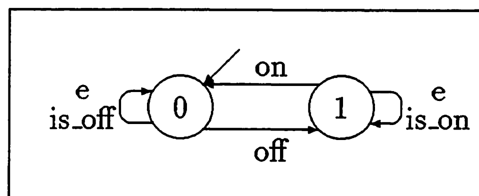


Graphe 1: le système de transitions tâche

2.2.2 La CPU

L'hypothèse la plus générale pour la CPU consite à la supposer interruptible, et indépendante des priorités des tâches. Le système de transitions d'une CPU interruptible pour N tâches s'obtient aisément par produit de synchronisation de N systèmes de transition réprésentant chacun une CPU mono-tâche.
Considérons pour cela le système de transition et le système de synchronisation ci-contre (Exemple d'une CPU pour quatre tâches). Intuitivement, la CPU multi-tâches est une pile de N CPU mono-tâche :

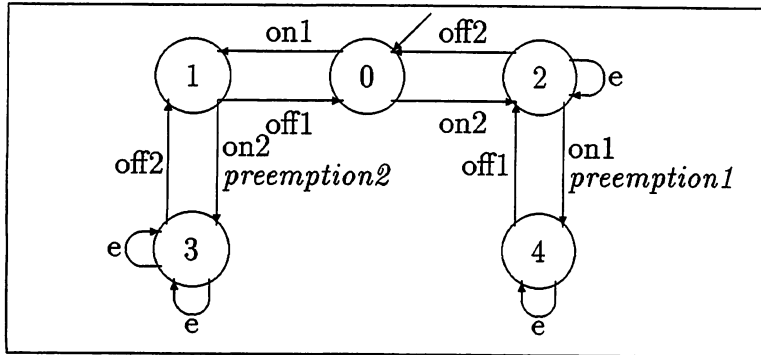
- la $(p+1)^{eme}$ n'est accéssible que lorsque les p premières sont déja utilisées;
- la $(p-1)^{eme}$ n'est libérable que lorsque les $n-p$ dernières sont déja libérées



Graphe 2: u ne CPU

```
synchronization_system  cpu4
 <width=4; list =(cpu,cpu,cpu,cpu)>;
( on     . e      . e      . e   );
( off    . e      . e      . e   );
( is_off . on     . e      . e   );
( is_off . off    . e      . e   );
( is_off . is_off . on     . e   );
( is_off . is_off . off    . e   );
( is_off . is_off . is_off . on  );
( is_off . is_off . is_off . off );
( e      . e      . e      . e   );
```

Le système de transition MEC dans le cas de deux tâches



Graphe 3: une CPU, pour 2 tâches, interruptible

Le produit de synchronisation obtenu pour deux tâches est représenté ci-contre.
Les marques de transitions *preemption1, preemption2* permettent de différencier les deux interruptions possibles. Elles seront utilisées pour restreindre les calculs en fonction des hypothèses sur le comportement de la CPU.

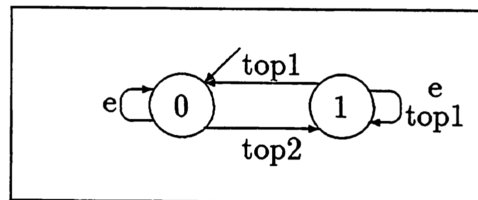
Le système de transition MEC dans le cas de N tâches Il est aisé de se convaincre que le produit de synchronisation obtenu pour N tâches est de $\sum_{p=0}^{N} \frac{N!}{p!}$ états et de $2\left(\sum_{p=0}^{N} \frac{N!}{p!}\right) - 1$ transitions, et qu'il représente effectivement le comportement d'une CPU interruptible par toutes les tâches, indépendemment de leur niveau de priorité.

2.2.3 L'horloge

L'horloge délivre des tops destinés à activer les différentes tâches. Nous avons choisi de modéliser notre hypothèse sur l'ordre des périodicités des différentes tâches dans le comportement de l'horloge. Celle-ci est donc légèrement plus complexe qu'un simple emetteur non déterministe de tops.

Le système de transition de l'horloge dans le cas de 2 tâches

Le système de transitions ci-contre décrit des comportements dans lesquels il est impossible d'avoir deux `top2` consécutifs, ce qui exprime bien le fait que la période d'activation de la tâche 1 est inférieure à celle de la tâche 2.



Graphe 4: une horloge pour 2 tâches

Le système de transition MEC de l'horloge pour N tâches
Nous avons choisi cette modélisation pour deux tâches car elle se généralise pour N tâches par produit de synchronisation. En effet le système de transition précédent modélise la contrainte *deux* `top2` *sont séparés par au moins un* `top1`. Nous pouvons donc obtenir une horloge pour 3 tâches (resp. N) par synchronisation de trois (resp $N(N-1) \div 2$) systèmes représentant respectivement les relations : *deux* `top2` *séparés par* `top1`, *deux* `top3` *séparés par* `top1` et *deux* `top3` *séparés par* `top2`, (resp *deux* `topi` *séparés par* `topj` pour i > j).

```
synchronization_system  horloge4 < width = 6 ; list =
(horloge2,horloge2,horloge2,horloge2,horloge2,horloge2) > ;
( top1 . top1 . top1 . e . e . e ); renomme top1
( top2 . e . e . top1 . top1 . e ); renomme top2
( e . top2 . e . top2 . e . top1 ); renomme top3
( e . e . top2 . e . top2 . top2 ). renomme top4
```

Il est malaisé de se convaincre que le produit de synchronisation obtenu est de $N!$ états et de $\sum_{p=1}^{N} \frac{N!}{p}$ transitions, et qu'il représente effectivement le comportement d'une horloge qui émet des tops tels que les périodes soient ordonnées en fonction du numéro des tâches.

3 Le comportement global du système

L'étude qui suit est effectuée dans le cas de deux tâches, mais tous les calculs réalisés peuvent être généralisés (avec une plus ou moins grande facilité) au cas de N tâches. Il suffit d'utiliser N tâches, une CPU pour N tâches et une horloge pour N tâches.

3.1 Le comportement dans le cas de deux tâches

Considérons le système de synchronisation suivant :

```
synchronization_system rma2 <width =4; list =
(tache,tache,cpu2,horloge2)>;
(e         . cpu_off   . off2 . e    );
(e         . cpu_on    . on2  . e    );
(cpu_off   . e         . off1 . e    );
(cpu_on    . e         . on1  . e    );
(my_top    . other_top . e    . top1 );
(other_top . my_top    . e    . top2 ).
```

medskipet les calculs MEC suivants :

```
sync(rma2,rma2);
dts(rma2);
```

Le produit de synchronisation obtenu par MEC comprend 34 états et 54 transitions. Nous pourrions évidemment faire les calculs classiques de blocage et de livelocks sur ce graphe, mais ces calculs ne seront fait que dans un cadre plus général, qui permettra la comparaison des résultats.

4 Comparaison des différentes hypothèses

4.1 L'approche

Les utilisateurs de MEC préférent en général calculer autant de systèmes de transitions qu'ils ont de cas différents à traiter. La comparaison des différentes hypothèses doit alors se faire à l'aide d'outil manipulant des équivalences de systèmes. Cette méthode est assez mal

adaptée à notre problème, car nos comparaisons d'hypothèses portent essentiellement sur des reconnaissances de parties communes. C'est pourquoi nous avons choisi d'utiliser une fonctionnalité de MEC, qui jusqu'alors avait été peu utilisée, à savoir que toutes les fonctions MEC ont la possiblité de ne s'appliquer qu' à un sous-système de transitions d'un système de transition donné. En contre-partie, cela nécessite l'introduction de nombreuses notations pour les marques calculées.

4.2 Les calculs

Les notations associées aux différentes hypothèses étudiées sont les suivantes :

- les marques d'états commencent par E;
- les marques de transitions commencent par T;
- La suite du label est composé d'un élément dans chacun des trois groupes suivants :
  1. c0 pour une cpu non interruptible;
  2. ci pour une cpu interruptible par toutes les tâches;
  3. c1 pour une cpu interruptible par la tâche 1;
  4. c2 pour une cpu interruptible par la tâche 2;

  1. p0 lorsqu'aucune tâche n'est prioritaire;
  2. p1 lorsque la tâche 1 est prioritaire
  3. p2 lorsque la tâche 2 est prioritaire

  1. t0 lorsqu'aucune période de tâche n'est fixée;
  2. t1 lorsque les périodes des tâches sont ordonnées par numéro de tâche;
- Exemple de marque : Ec1p2t1

Ayant choisi d'intégrer la contrainte sur les périodes dans le système de transitions qui modélise l'horloge, nous obtenons douze hypothèses, pour lesquelles nous calculons les douze graphes de comportements comme douze sous-graphes du produit de synchronisation initial, ceci à l'aide des douze restrictions sur les transitions suivantes :

```
Tok := * \/ rsrc(initial);
Tok := *  - bloquee[2] ;
Tok := * - (preemption1[3] \/ preemption2[3]) ;
Tok := (*  - (preemption1[3] \/ preemption2[3]))  - bloquee[1] ;
Tok := (*  - (preemption1[3] \/ preemption2[3]))  - bloquee[2] ;
Tok := *  - preemption2[3];
Tok := (* - preemption2[3]) - bloquee[1] ;
Tok := (* - preemption2[3]) - bloquee[2] ;
Tok := *  - preemption1[3] ;
Tok := (* - preemption1[3])- bloquee[1] ;
Tok := (* - preemption1[3]) - bloquee[2] ;
```

Les douze sous-graphes s'obtiennent alors par les calculs suivants (Exemple du cas cip1t1) qui correspond pour mémoire à:

- cpu interruptible par toutes les tâches;
- tâche 1 prioritaire;
- période 1 > période 2

```
\* le sous graphe *\
Ecip1t1 := reach(initial,Tok);
Tcip1t1 := (rtgt(Ecip1t1) /\ rsrc(Ecip1t1)) /\ Tok;
```

Ces sous-graphes peuvent alors îre analysés avec MEC comme tout système de transitions. Les notations associées aux différentes prioriétes vérifiées sont les suivantes :

- b1 lorsque la tâche 1 est bloquée;
- b2 lorsque la tâche 2 est bloquée;
- d0 pour les blocages simples;
- d1 pour les blocages inévitables;
- d2 pour les blocages créés par rapport au modèle initial;
- d3 pour les blocages supprimés par rapport au modèle initial;
- dt pour les blocages temporels;
- i1 lorsque la tâche 1 est interrompue;
- i2 lorsque la tâche 2 est interrompue;
- t1 pour la première contrainte temporelle;
- t2 pour la deuxième contrainte temporelle;
- Exemple : Ec1p2t1_i1

Et la série de calculs effectuée pour les douze sous-graphes (Exemple du cas initial cip0t1):

```
\* les marques d'etats *\
Ecip0t1_d0 := Ecip0t1- src(Tcip0t1);
Ecip0t1_d1 := unavoidable(Tcip0t1,Ecip0t1_d0) /\ Ecip0t1;
Ecip0t1_d2 := Ecip0t1_d1 - Ecip0t1_d1;
Ecip0t1_d3 := Ecip0t1_d1 - Ecip0t1_d1;

\* les marques de transitions *\
Tcip0t1_b1 := bloquee[1] /\ Tcip0t1;
```

```
Tcip0t1_b2 := bloquee[2] /\ Tcip0t1;
Tcip0t1_i1 := suspendue[1] /\ Tcip0t1;
Tcip0t1_i2 := suspendue[2] /\ Tcip0t1;
Tcip0t1_d1 := (rsrc(Ecip0t1-Ecip0t1_d1)/\Tcip0t1) -

              (rtgt(Ecip0t1-Ecip0t1_d1)/\Tcip0t1);
Tcip0t1_dt := Tcip0t1_d1 /\ (top1[4] \/ top2[4]);
```

4.3 L'interprétation des résultats

Nous constatons que pour les douze hypothèses, les ensembles :

- `Tc?p?t1_d1` qui représente les transitions qui passent de l'ensemble des états non bloquants à l'ensemble des états inévitablement bloquants;
- `Tc?p?t1_dt` qui représente les transitions `Tc?p?t1_d1` qui sont des `top?`;

sont égaux et non nuls. Nous pouvons ainsi conclure que tous les blocages sont provoqués par un top d'horloge intempestif, d'ou leur nom de blocages temporels.

Nous constatons également que l'ensemble :

- `Ec?p?t1_d2` qui représente les états bloquants qui ne sont pas des états bloquants dans le modèle de référence.

est non vide lorsque la tâche prioritaire ne peut interrompre la tâche non prioritaire (Ce sont les cas `c0p1t1`, `c0p2t1`, `c1p2t1` et `c2p1t1`).

5 Synthèse du contrôleur temporel

Le but de cette partie est de déterminer un ensemble de relations temporelles sur les périodes et les durées des tâches qui éliminent les blocages temporels. Pour cela, nous utilisons la méthode décrite dans l'introduction 2.

5.1 Les contraintes temporelles

Nous allons nous intéresser à des contraintes temporelles sur les durées d'occupation de la CPU par les différentes tâches, et déterminer si elles sont suffisantes pour garantir un fonctionnement sans blocage du système global.
Soient $d1$ (resp $d2$) le temps d'occupation de la CPU par la tâche 1 (resp 2); et $T1$ (resp $T2$) le temps entre deux `top1` (resp `top2`). Les deux contraintes que nous allons étudier sont les suivantes :

1. $d1 + d2 < T1$
2. $d1\,T2 + d2\,T1 < T1\,T2$

Il faut noter que du fait que la période des top1 est inférieure à celle des top2, nous avons forcément $T1 < T2$.

D'une manière générale, le calcul consiste à poser sur chacune des transitions l'ensemble des marques ij qui indiquent qu'elle peut avoir lieu i (resp j) instants (dans une base de temps symbolique) après le dernier top1 (resp. top2). Cette base de temps symbolique est déterminée par la donnée des ensembles de transitions suivants :

- les transitions gratuites (dont la durée est nulle) vis à vis des top1;
- les transitions gratuites vis à vis des top2;
- les transitions coûteuses (dont la durée est non nulle) vis à vis des top1;
- les transitions coûteuses vis à vis des top2;
- les transitions qui réinitialisent la base de temps vis à vis des top1;
- les transitions qui réinitialisent la base de temps vis à vis des top2;

Ces ensembles peuvent être vus comme les paramètres de fonctions de transitions qui décrivent les évolutions des bases de temps symboliques. Ces fonctions d'évolution du temps peuvent ensuite être décrites sous forme de systèmes de transitions (ancienne approche), ou bien sous forme de points fixes (notre approche).

6 La contrainte `d1 + d2 < T1`

6.1 L'approche

Modéliser le fait que $d1 + d2 < T1$ est vérifiée consiste à retirer du graphe les tops d'horloge qui ne respectent pas cette contrainte (intuitivement, deux tops identiques consécutifs doivent être séparés par un temps d'occupation de la CPU, suffisant pour que la tâche1 et la tâche2, si elles le désirent ne soient pas perturbées).
Du fait, que notre modélisation du temps est symbolique, la relation $d1+d2 < T1$ n'implique pas la relation $d1+d2 < T2$. Il est donc nécessaire de tenir compte des deux contraintes dans notre calcul. Pour ces contraintes, il suffit de comptabiliser les libérations (les actions off) de la CPU, qui indiquent que la tâche a occupé la CPU pendant le temps d correspondant. Les ensembles de transitions utiles se réduisent donc aux suivants :

- les transitions gratuites;
- les transitions coûteuses;
- les transitions qui réinitialisent la base de temps vis à vis des top1;
- les transitions qui réinitialisent la base de temps vis à vis des top2;

Notre temps symbolique est donc pour chacun des tops, une horloge à trois valeurs (0, 1 et 2) ou la mesure du temps est rythmée par les occurrences de (off1, off2).
La dernière ligne du tableau ci-contre représente le cas ou les deux tâches sont dans un état d'attente libre, dans lesquels il est normal que le temps puisse s'écouler librement.

	raz1	raz2	cout	gratuit
00			11	00
01			12	01
02		00	12	02
10			21	10
11			22	11
12		10	22	12
20	00		21	20
21	01		22	21
22	02	20	22	22
depart	02	20	22	22

6.2 La fonction point fixe

La fonction de transition précédente définit un point fixe que l'on peut exprimer dans la syntaxe MEC. Par exemple, l'équation de l'attente symbolique 12 est :

```
function T22attente(Epar:state ; depart:state ; Tpar:trans;
               raz1:trans; raz2:trans; cout:trans; gratuit:trans)
    return attente:trans;
begin
...
attente12 = ((({} \/
             {}) \/
             (rsrc(tgt(attente01 \/ attente02)) /\ cout)) \/
             (rsrc(tgt(attente12)) /\ gratuit))
             /\ Tpar;
...
attente = ((((((((attente00 \/ attente01) \/ attente02) \/
           attente10) \/ attente11) \/ attente12) \/
           attente20) \/ attente21) \/ attente22)
end.
```

Les paramètres Epar et Tpar permettent de restreindre le calcul à un sous graphe. Cela sera utilisé pour étudier cette contrainte pour les douze hypothèses de la section précédente.

6.3 Le produit de synchronisation

Le système de synchronisation, ainsi que le produit de synchronisation, sont identiques à ceux de la section 3.1. Nous calculons initialement les paramètres utiles à la fonction point fixe.

```
sync(rma2,rmatT2);
dts(rmatT2);
repos := repos[1] /\ repos[2];
on := !label[3] = 'on*';
```

```
off := !label[3] = 'off*';
top1 := !label[4] = 'top1';
top2 := !label[4] = 'top2';
```

6.4 Les calculs

Les douze sous-graphes sont calculés comme lors de la comparaison des hypothèses; puis pour chacun des sous-graphes nous calculons (Exemple du cas `Ecip0t1`) :

```
\* les transitions possibles avec les contraintes d1+d2<top1 et
d1+d2<top2 *\
Tok := T22attente(Ecip0t1,repos,Tcip0t1,top1,top2,off,on);

\* le sous graphe possibles avec les contraintes d1+d2<top1 et
d1+d2<top2 *\
Ecip0t1_t1 := reach(initial,Tok);
Tcip0t1_t1 := (rtgt(Ecip0t1_t1) /\ rsrc(Ecip0t1_t1)) /\ Tok;

\* les marques d'etats *\
Ecip0t1_d0 := Ecip0t1_t1 - src(Tcip0t1_t1);
Ecip0t1_d1 := unavoidable(Tcip0t1_t1,Ecip0t1_d0) /\ Ecip0t1_t1;

\* les marques de transitions *\
Tcip0t1_b1 := bloquee[1] /\ Tcip0t1_t1;
Tcip0t1_b2 := bloquee[2] /\ Tcip0t1_t1;
Tcip0t1_i1 := suspendue[1] /\ Tcip0t1_t1;
Tcip0t1_i2 := suspendue[2] /\ Tcip0t1_t1;
```

6.5 Les résultats

Nous constatons que l'ensemble :

- `Ec?p?t1_d0` qui représente les états bloquants

est vide pour huit des douze hypothèses. Les quatres cas pour lesquels cet ensemble est non vide sont :

- `Ec0p1t1`, `Ec0p2t1`, `Ec1p2t1` et `Ec2p1t1`

Pour ces quatre cas, la tâche prioritaire ne peut interrompre l'autre, alors que pour les huit autre cas, cela est toujours possible.

6.6 Propriété

La contrainte $d1 + d2 < T1$ est suffisante pour garantir un fonctionnement sans blocage si la tâche prioritaire peut interrompre l'autre tâche.

6.7 Remarque

Nous avons également traduit la fonction d'évolution des bases de temps symboliques associée à cette contrainte en un système de transitions. Nous l'avons intégré dans un nouveau système de synchronisation, afin de calculer un nouveau produit de synchronisation.
Nous avons obtenu pour le graphe le plus général, `Ecip0t1` :

- 30 états ;
- 41 transitions ;
- 0 état bloquant.

Nous aurions pu calculer pour chacun des douze sous-graphes les états bloquants. Nous ne l'avons pas fait, mais nous pouvons montrer qu'il ne peut y en avoir.
Nous remarquons que sur cet exemple, le gain en taille n'est pas significatif.

7 La contrainte $d1\,T2 + d2\,T1 < T1\,T2$

7.1 L'approche

Nous écrivons la contrainte $d1\,T2 + d2\,T1 < T1\,T2$ sous la forme $d1 + \frac{T1}{T2}d2 < T1$. En effet, modéliser cette dernière consiste à retirer les tops d'horloge qui ne la respectent pas (intuitivement, deux tops identiques consécutifs doivent être séparés par un temps d'occupation de la CPU, suffisant pour que la tâche1 et un pourcentage de la tâche2, si elles le désirent ne soient pas perturbées).
Du fait, que notre modélisation du temps est symbolique, la relation $d1 + \frac{T1}{T2}d2 < T1$ n'implique pas la relation $\frac{T2}{T1}d1 + d2 < T2$. Il est donc nécessaire de tenir compte de ces deux contraintes dans notre calcul. Pour ces contraintes, il suffit de comptabiliser les prises (les actions on) et les libérations (les actions off) de la CPU, qui indique que la tâche a commencé, puis terminé d'occuper la CPU pendant le temps d correspondant. Les ensembles de transitions utiles se réduisent donc aux suivants :

- les transitions gratuites pour les deux tâches;
- les transitions coûteuses pour les deux tâches;
- les transitions coûteuses pour la tâche 1 seulement;
- les transitions coûteuses pour la tâche 2 seulement;
- les transitions qui réinitialisent la base de temps vis à vis des top1;
- les transitions qui réinitialisent la base de temps vis à vis des top2;

Notre temps symbolique est donc :

- pour top1, une horloge à trois valeurs (0, 1 et 2) et la mesure du temps est rythmée par les occurrences de (off1, on2, off2);
- pour top2, une horloge à cinq valeurs (0, 1, 2, 3 et 4) et la mesure du temps est rythmée par les occurrences de (on1, off1, off2);

La dernière ligne du tableau ci-contre représente le cas ou les deux tâches sont dans un état d'attente libre, dans lesquels il est normal que le temps puisse s'écouler librement.

	raz1	raz2	cout1	cout2	cout	gratuit
00			10	01	11	00
01			11	02	12	01
02			12	03	13	02
03			13	04	14	03
04		00	14	04	14	04
10			20	11	21	10
11			21	12	22	11
12			22	13	23	12
13			23	14	24	13
14		10	24	14	24	14
20	00		20	21	21	20
21	01		21	22	22	21
22	02		22	23	23	22
23	03		23	24	24	23
24	04	20	24	24	24	24
depart	04	20	24	24	24	24

7.2 La fonction point fixe

Comme nous l'avons déjà vu, une telle fonction de transition définie un point fixe que l'on peut exprimer dans la syntaxe MEC.

```
function T24attente
  (Epar:state ; depart:state ; Tpar:trans; raz1:trans; raz2:trans;
   cout1:trans; cout2:trans; cout:trans; gratuit:trans)
    return attente:trans;
begin
...
attente23 = (((((({} \/
            {}) \/
            (rsrc(tgt(attente13\/attente23)) /\ cout1)) \/
            (rsrc(tgt(attente22)) /\ cout2)) \/
            (rsrc(tgt(attente12\/attente22)) /\ cout)) \/
            (rsrc(tgt(attente23)) /\ gratuit))
            /\ Tpar;
...
attente = ((((((((((((
(attente00 \/ attente01) \/ attente02) \/ attente03) \/  attente04) \/
 attente10) \/ attente11) \/ attente12) \/ attente13) \/ attente14) \/
 attente20) \/ attente21) \/ attente22) \/ attente23) \/ attente24)
end.
```

Les paramètres `Epar` et `Tpar` ont la même sémantique que précédemment.

7.3 Le produit de synchronisation

Le système de synchronisation, ainsi que le produit de synchronisation, sont identiques à ceux de la section 3.1. Nous calculons initialement les paramètres de la fonction point fixe.

```
repos := repos[1] /\ repos[2];
on1 := !label[3] = 'on1';
on2 := !label[3] = 'on2';
off1 := !label[3] = 'off1';
off2 := !label[3] = 'off2';
top1 := !label[4] = 'top1';
top2 := !label[4] = 'top2';
```

7.4 Les calculs

Ils sont identiques à ceux de la première contrainte, seul la fonction point fixe utilisée diffère. La ligne :

```
Tok := T22attente(Ecip0t1,repos,Tcip0t1,top1,top2,off,on);
```

est remplacée par :

```
Tok :=
T24attente(Ecip0t1,repos,Tcip0t1,top1,top2,on2,on1,(off1\/off2),{});
```

7.5 Les résultats

L'ensemble `Ec?p?t1_d0` qui représente les états bloquants est toujours non vide.

7.6 Propriété

La contrainte $d1 + \frac{T1}{T2}d2 < T1$ est insuffisante pour garantir un fonctionnement sans blocage.

7.7 Remarque

Nous avons également traduit la fonction de transition associée à cette contrainte en un système de transitions. Nous avons obtenu pour le système de transitions le plus général, `Ecip0t1`, un graphe de **105 états** et de **144 transitions**.
Nous constatons sur cet exemple, que le gain en taille peut être très significatif.

8 La modélisation des contraintes temporelles

Question 1: Pourquoi la contrainte $d1+d2 < T1$, par exemple, nécessite t-elle la modélisation de la contrainte $d1 + d2 < T2$, alors que la contrainte $T1 < T2$ est modélisée par l'horloge ?

Réponse : L'horloge modélise une contrainte légèrement différente, avec laquelle la transitivité ne peut s'appliquer.
Question 2: Pourquoi les deux contraintes $d1+d2 < T1$ et $d1+d2 < T2$ sont elles modélisées par une seule fonction de transition, alors qu'une fonction de transition représentant une contrainte paramétrée $d1 + d2 < T$ utilisée dans les deux contextes suivants :

1. $d1 + d2 < T$ avec $T = T1$
2. $d1 + d2 < T$ avec $T = T2$, restreint au sous-graphe obtenu.

semble suffire ?
Réponse : Cela semble en effet suffisant, mais MEC ne dispose pas aujourd'hui de toutes les fonctions nécéssaires pour traiter cette question.

9 Conclusion et perspectives

Les résultats présentés dans cet article ne sont pas nouveaux. En effet, tous les spécialistes des problèmes temps réels les connaissent depuis longtemps et ils ne servent ici que d'exemples pour illustrer la démarche employée.
Les techniques présentées dans cet article semblent bien adaptées et efficaces pour traiter de la synthèse pour des problèmes temps réels. Cependant, afin qu'elles puissent réellement être mise en oeuvre dans des problèmes industriels, il est nécessaire d'ajouter à MEC un certain nombre de fonctionnalités.
Nos recherches actuelles sont donc orientées vers :

- Quelles sont les classes de contraintes temporelles pour lesquelles notre modélisation symbolique du temps est suffisante ?
- Quelles modifications faut-il apporter au modèle Arnold-Nivat pour traiter efficacement ces types de calculs ?
- Quelles sont les types de propriétés que l'on peut alors calculer ?

10 Remerciements

Merci à Jean Pierre Radoux qui m'a aiguillé sur ce problème, à Anne Dicky et André Arnold pour leurs nombreuses et fructueuses observations.

References

[Arn89] André Arnold. Mec: a system for constructing and analysing transition systems. In J. Sifakis, editor, *Automatic Verification of Finite State Systems*, pages 117–132. LNCS 407, 1989.

Vérification de protocoles : le point de vue d'un opérateur

Roland Groz - France Télécom CNET
CNET LAA/EIA/EVP
BP 40
F-22301 Lannion cedex (France)
groz@lannion.cnet.fr

Résumé de la présentation

L'intention de cette présentation est de mettre en évidence l'apport des méthodes formelles de vérification, mais surtout leurs limites actuelles, par rapport aux attentes des «spécifieurs en télécommunications».

Dans une première partie, qui porte plutôt à l'optimisme, on verra que des méthodes «formelles» comme Estelle et LDS ont pu être utilisées avec succès, y compris pour faire de la vérification, sur des protocoles réels et de taille significative.

Dans une deuxième partie, on analysera pourquoi les méthodes purement mathématiques (comme les systèmes de transitions) n'ont pu et ne peuvent avoir le même impact que des techniques comme Estelle, LDS et Lotos.

Une troisième partie, fondée sur l'expérience d'une douzaine de cas concrets de formalisation de protocoles traités au CNET dans des domaines divers, analysera les besoins actuels en spécification chez un opérateur de télécommunications. On verra comment les évolutions rapides de ce secteur font surgir des problèmes nouveaux, mal couverts par les techniques de vérification dont nous disposons. Ceci permettra d'esquisser quelques nouvelles directions de recherche soutenues actuellement par le CNET.

1. Méthodes «formelles» et vérification : situation actuelle au CNET

a) Le contexte

Le CNET (Centre National d'Études des Télécommunications) est le centre de recherches de l'opérateur de télécommunications France Télécom. Comme l'opérateur n'est pas impliqué dans la production de systèmes (qui sont achetés à des fournisseurs industriels), l'activité du CNET, hormis les recherches purement technologiques, se concentre sur les extrémités du cycle de développement de nouveaux systèmes :

- la phase amont, avec la spécification des systèmes, principalement au niveau de leurs interfaces
- la recette des équipements pour vérifier leur conformité aux spécifications.

L'importance de l'activité de spécification (plus de 1000 ingénieurs du CNET participent directement à cette activité), ainsi que le besoin de produire des spécifications cohérentes ont amené le CNET à s'intéresser, depuis une douzaine d'années, aux techniques de spécifications formelles et aux méthodes de vérification associées. [Groz 91]

b) Approche retenue

Les principales orientations choisies par le CNET ont été les suivantes.

- Des techniques formelles : vite apparues comme la base nécessaire pour maîtriser la complexité des protocoles.
- Des techniques normalisées : pour les protocoles, cela signifie les langages inclus dans le sigle FDT (Techniques de Description Formelle), normalisés à cette fin par l'ISO et l'UIT-T, à savoir Estelle, Lotos et LDS. En pratique, Lotos n'a encore jamais été utilisé par le CNET pour des applications réelles. Au delà de ces techniques de description de protocoles et services, il y a des langages plus ciblés, issus des mêmes organismes, qui ont pris une importance grandissante : ASN.1, TTCN, GDMO.
- Des outils logiciels, sans lesquels aucune technique ne trouve d'application pratique.

En ce qui concerne la mise au point de techniques de vérification et des outils associés, nous nous sommes appuyés sur des collaborations extérieures afin d'aider à mettre en valeur le savoir faire d'équipes universitaires ou industrielles. Nous avons ainsi par le passé collaboré avec des équipes universitaires françaises : le LAAS, pour l'outil OGIVE/D d'analyse de réseaux de Petri dans les années 80, puis l'IMAG pour la vérification de modèle avec l'outil Xésar [Richier 87]; nous avons promu et développé l'utilisation de la simulation aléatoire comme technique de validation [Jard 88] [Groz 85] [Cavalli 88]; nous avons participé à des projets européens comme Sedos Estelle Demonstrator [Ayache 89] du programme ESPRIT, et Specs [Specs 93] du programme RACE; enfin, nous avons assisté le développement d'outils par des partenaires industriels [Algayres 91].

Le développement des méthodes et des outils s'est bien sûr accompagné, au sein du CNET, d'un effort de dissémination de ces techniques par la formation et l'assistance (expertise apportée sur les problèmes applicatifs).

L'accroissement des capacités des méthodes de spécification et de vérification formelle se voit bien sur les jalons suivants, traités au CNET :

- en 1983, vérification d'un protocole d'exclusion mutuelle avec diverses techniques, dont l'outil OGIVE/D [Groz 85]. Taille : 200 lignes d'Estelle
- en 1987, vérification du protocole T70 du CCITT (Télétex couche transport) avec Véda et Xésar. Taille : 400 lignes d'Estelle.
- en 1989, spécification du protocole LAPD du RNIS. Taille : 8000 lignes d'Estelle. Pas d'objectif de vérification, mais simulation et génération de tests.
- en 1992 et 1993 : on s'intéresse à des protocoles ATM comme FRP et SSCOP, qu'on vérifie avec l'outil Vésar. Taille : de 1000 à 2000 lignes d'Estelle.

c) Situation actuelle : des perspectives encourageantes

Actuellement, un certain nombre de protocoles tout à faits significatifs (c'est à dire «gros» et représentatifs de leur domaine) ont été formalisés, en Estelle ou en LDS: T70, LAPD, Protocole D niveau 3 (appel de base), RTSE, P1, FRP, SSCOP, MAP (du GSM)...

De nouvelles actions sont en cours pour formaliser systématiquement certaines classes de spécifications, en particulier dans le domaine des compléments de service pour le téléphone numérique (RNIS).

Cependant, ces actions sont moins souvent motivées par le besoin de vérification que par celui

d'exploiter ces spécifications pour faciliter les comparaisons entre versions, et surtout pour engendrer des tests de conformité. Les résultats obtenus avec l'outil TVEDA [Phalippou 90] permettent d'envisager raisonnablement d'automatiser à plus de 80% la production des tests.

Ce qui a permis de pousser la formalisation n'est donc pas le besoin de vérification, qui apparaît comme secondaire dans la plupart des cas [Rouger 89] [Phalippou 90]. On verra plus en détail comment le type de vérification fait actuellement est relativement mal adapté aux besoins des équipes d'un centre dédié aux télécommunications (cf les parties suivantes).

2. Pourquoi les méthodes purement mathématiques rencontrent peu d'écho

Le peu d'impact des méthodes mathématiques dans «l'industrie» du logiciel et de la téléphonie est une tarte à la crème des discussions au cours des colloques consacrés aux modèles formels, comme BMW.

Je sacrifierai donc à cette habitude, mais en essayant d'appuyer mon analyse sur l'expérience acquise au CNET dans le domaine de la vérification des protocoles.

En particulier, l'argument habituel selon lequel les méthodes formelles ne peuvent traiter que des problèmes de taille restreinte, sans intérêt pratique, ne me semble plus valide.

Afin de laisser aux participants à BMW la primeur de cette analyse, on ne trouvera ici que les titres principaux.

a) Le contrôle n'est qu'un aspect, difficile à démêler

b) Elles s'appuient sur des experts en modélisation

c) Elles ne traitent pas les bons problèmes

3. Problèmes de vérification : besoins actuels mal couverts

Les applications de formalisation passées ou en cours au CNET ont fait apparaître les limites des méthodes de vérification par rapport aux problèmes traités et aux attentes (et tout simplement aux besoins) des spécifieurs. Cette analyse se fonde en particulier sur les applications suivantes : LAPD (du RNIS) et SSCOP (protocole pour l'ATM) pour les couches basses (protocole RNIS), P1 (messagerie X.400) et la gestion (administration) de réseaux pour les couches hautes, problèmes rencontrés sur les services RNIS.

a) Les problèmes connus, mais complexes, et malheureusement fréquents (voire omniprésents)

La plupart des protocoles, et c'est le cas du LAPD (mais le protocole «académique» INRES en est un exemple réduit également) font intervenir des mécanismes qui induisent une grande complexité de la vérification, mais dont la validation est essentielle si on veut prétendre traiter réellement le protocole. Ce sont en particulier les mécanismes suivants (cas du LAPD) :

- file non bornée de messages; dans le LAPD, une entité peut stocker un nombre de messages avant de se décider à les transmettre; ces effets de tampons deviennent prépondérants dans des domaines comme l'ATM

- fenêtre d'émission; au cœur du LAPD, il y a la fenêtre du protocole HDLC. Pour valider les mécanismes de retransmission, il faut absolument représenter la possibilité d'avoir un certain nombre de messages en transit; dans le cas de SSCOP, comme on s'intéresse à des modes de retransmission sélective (par désignation d'ensembles de messages perdus, il faut que des configurations non triviales d'ensemble soient représentées dans le modèle formel; on ne peut donc se contenter de représenter deux messages en transit
- temporisations; celles-ci interfèrent avec d'autres mécanismes comme la retransmission, mais de nombreux modèles formels produisent des résultats inexploitables (erreurs parasites par dizaines ou milliers) lorsque les aspects temporels ne sont pas pris en compte
- traitements d'erreurs, comportements «inopportuns» (selon la terminologie du test dans la norme IS 9646); souvent décrits de façon ambiguë, ils sont source d'erreur, mais comme ils peuvent intervenir dans des phases très diverses, ils augmentent considérablement la complexité du modèle formel, ce qui les fait souvent omettre de celui-ci.

b) La formalisation n'est qu'une modélisation : le spécifieur est «trahi»

Lorsqu'on lit la spécification informelle d'un protocole comme le LAPD, on se rend compte que plus du tiers de cette spécification est consacrée à spécifier un point clé de tout protocole, superbement ignoré de toutes les méthodes formelles : le codage bit par bit des messages échangés. C'est un point-clé parce que sans accord sur ce codage, il ne peut y avoir de communication entre machines hétérogènes, donc pas de «protocole». Ce que le modèle formel va représenter, c'est la partie comportementale, appelée «procédures» dans le jargon des spécifications de protocoles. Dans le cas d'un protocole comme P1 (entre deux agents de transferts de message X.400), tout le protocole n'est décrit que par les champs de ses messages (principalement les différents champs de l'en-tête comme «expéditeur», «destinataire» etc), les procédures étant plus ou moins réduites à un commentaire sur l'interprétation à donner à chaque champ.

Le premier obstacle que cela soulève, c'est que la spécification n'est qu'une modélisation partielle, qui ne peut prétendre remplacer la spécification informelle. La formalisation ne devient donc qu'un choix particulier, dont se désintéresse le responsable de la spécification.

Par ailleurs, nous avons actuellement un problème concret de vérification de cohérence entre des versions successives du RNIS. Le terminal téléphonique acheté par un client à un moment donné doit rester compatible avec les évolutions futures du réseau (pour poser le problème vu du client !). Ceci suppose qu'on soit capable de vérifier complètement la mise en relation de deux spécifications de protocoles. L'expérience montre que dans ce cas, près de 90% des problèmes de cohérence entre les spécifications résident dans les codages des informations échangées.

c) Couches hautes : les besoins se déplacent vers le fonctionnel et les données

Comme on l'a vu, dans un protocole comme P1, l'essentiel de la spécification est dans les données. Pour valider des protocoles de la couche applicative, on doit s'intéresser aux fonctions qu'ils offrent, et à la cohérence des données échangées; les automates de contrôle sont en général réduits, ou bien correspondent à des traitements séquentiels; ainsi pour P1, le contrôle consiste essentiellement en une descente hiérarchique de la structure des champs du message.

Il est d'ailleurs significatif que pour ce type de protocole, l'effort de «semi-formalisation»

qu'on trouve dans les spécifications porte sur les données. Alors que dans les spécifications des couches basses on trouve des tables d'états qui sont une aubaine pour le spécialiste des méthodes à base d'automates, dans les couches hautes on a des formalismes comme ASN.1, qui ne décrit que les données et leur codage, ou GDMO qui définit les attributs des objets du réseau administré, mais pas leur comportement. D'ailleurs, après investigation, les propositions de formalisation des comportements en GDMO se sont écartées d'Estelle, LDS et Lotos pour s'orienter vers Z.

d) Les besoins sont en recherche de cohérence plus qu'en vérification

Le principal problème des spécifieurs en télécommunications n'est pas une vérification fine de la cohérence du comportement dynamique d'un protocole. Une des raisons essentielles est qu'il devient très rare qu'un nouveau protocole soit créé ex nihilo. De même que le métier d'opérateur évolue de plus en plus vers l'assemblage de systèmes (en garantissant leur interfonctionnement), de même les spécifications de protocoles sont souvent des variations sur un thème déjà connu, ou l'intégration (de type fusion) entre d'autres spécifications. Ceci induit des besoins différents de la vérification classique.

- cohérence incrémentale : lors de l'ajout d'un paragraphe, retrouver ceux avec lesquels il interfère
- cohérence entre différentes versions d'un même protocole (cf supra)
- cohérence entre des services complémentaires; c'est le problème dit de «l'interaction de services», très à la mode en téléphonie : par exemple, que se passe-t-il si un abonné B renvoie sa ligne vers celle d'un abonné C, et qu'un abonné A, qui est dans la liste noire de C, appelle B ?

Dans la plupart des cas, on s'intéresse moins à une vérification fine qu'à une simple détection d'incohérence. Ne serait-ce, par exemple, que dresser la table des services dont l'intersection peut avoir un sens, ou donner lieu à des ambiguïtés.

Les techniques de vérification qui permettraient de couvrir ces besoins de cohérence assez faible nous font cruellement défaut, alors que nous nous sentons maintenant mieux équipés pour trouver des blocages subtils dans des spécifications complexes !

4. Nouvelles orientations de recherche

Bien entendu, il semble opportun de maintenir l'effort déjà fait avec les techniques de vérification de modèles, de réductions de systèmes de transitions etc pour améliorer et enrichir les capacités des outils existants. Néanmoins, le CNET est particulièrement intéressé par de nouvelles directions pour répondre à des besoins actuellement moins bien couverts. On peut en citer deux en particulier.

D'abord, si l'on considère que la partie «contrôle» est bien couverte, la validation des aspects «données» et corrélativement des aspects «fonctionnels» nécessite des techniques différentes, issues des travaux sur la spécification formelle des logiciels «séquentiels». Nous menons actuellement des études et des expérimentations sur des techniques comme Z, B, et l'utilisation du lambda-calcul typé pour la construction de programmes.

Ensuite, pour mieux couvrir les besoins en vérification de cohérence de spécifications partiellement formalisées, nous collaborons avec des équipes universitaires pour l'utilisation de techni-

ques issues du traitement de la langue naturelle, de la gestion documentaire (avec des enrichissements sémantiques), ou de la formalisation des processus de conception et de spécification.

5. Bibliographie

[Algayres 91] B. Algayres, V. Coelho, L. Doldi, H. Garavel, Y. Lejeune, D. Pilaud, C. Rodriguez: *VÉSAR: un outil pour la spécification et la vérification formelle de protocoles*; Colloque Francophone sur l'Ingéniérie des Protocoles (CFIP'91), Pau, septembre 1991; actes publiés chez Hermès, O. Rafiq réd.

[Ayache 89] J.M. Ayache, J. Berrocal, S. Budkowski, M. Diaz, J. Dufau, A.M. Druilhe, N. Echevarria, R. Groz, M. Huybrechts: *Presentation of the Sedos Estelle Demonstrator project*; The Formal Description Technique Estelle,North-Holland (Elsevier), 1989, pp 423-437.

[Cavalli 88] A.R. Cavalli, E. Paul: *Exhaustive analysis and simulation for distributed systems, both sides of the same coin*; Distributed Computing (1988) 2, pp 213-225.

[Groz 85] R.Groz, C. Jard, C. Lassudrie: *Attacking a Complex Distributed Algorithm from Different Sides: an Experience with Complementary Validation Tools*; Computer Networks and ISDN Systems, 10 (1985), pp 245-257.

[Groz 91] R. Groz, M. Phalippou: *L'ingénierie des protocoles au CNET*; Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'91), Pau, septembre 1991; Hermès 1991, O. Rafiq réd., pp 3-20.

[ISO 89e] ISO/TC97/SC21: *Estelle: A Formal Description Technique based on an Extended State Transition Model*; ISO IS 9074, 1989.

[ISO 89l] ISO/TC97/SC21: *Lotos: A Formal Description Technique based on the Temporal Ordering Of Observational Behaviour*; ISO IS 8807, 1989.

[Jard 88] C. Jard, R. Groz, J.F. Monin: *Development of VEDA: a prototyping tool for distributed algorithms*; IEEE Transactions on Software Engineering, 14 (3), mars 1988, pp 339-352.

[Phalippou 88] M. Phalippou, R. Groz: *Using Estelle for verification. An experience with the T.70 teletex transport protocol*; FORTE'88, Stirling (Écosse), Kenneth Turner (réd), North-Holland, septembre 1988.

[Phalippou 90] M. Phalippou, R. Groz, *Evaluation of an empirical approach for computer-aided test cases generation*, proceedings of the 3rd International Workshop on Protocol Test Systems, Washington,

October 1990.

[Richier 87] J.L. Richier, C. Rodriguez, J. Sifakis, J. Voiron: *Verification in XESAR of the sliding window protocol*; Protocol Specification, Testing and Verification VII (1987), Zürich, Elsevier, H. Rudin réd.

[Rouger 89] A. Rouger, P. Combes, *Exhaustive validation and test generation in ELVIS*, proceedings of the 4th SDL forum, SDL'89 The Language at Work, North-Holland, 1989.

[Specs 93] The Specs Consortium: *Specification and Programming Environment for Communication Software;* Elsevier (1993) ISBN 0 444 89923 5, R. Reed, W. Bouma, J. Evans, M. Dauphin & M. Michel réd.

Specifying Features and Analysing Their Interactions in a LOTOS Environment[1]

M. Faci and L. Logrippo

University of Ottawa, Protocols Research Group,
Department of Computer Science, Ottawa, Ontario, Canada K1N 6N5
E-mail: {mfaci, luigi}@csi.uottawa.ca

Abstract. This paper presents an approach for specifying telephone features and analysing their interactions in a LOTOS environment. The approach is characterized by a flexible specification structure and an analysis method based on knowledge goals. Structurally, the specifications allow the integration of new features into existing ones by specifying each feature independently and composing its behaviour with the existing system. Analytically, the reasoning mechanism allows the specifier to analyse features, for the purpose of detecting their interactions, by defining knowledge goals and simulating the system to verify if they are reachable. A non reachable knowledge goal reveals the existence of a feature interaction, or a design error. We explain this approach by the use of two, now classical, examples of feature interactions, namely Call Waiting & Three Way Calling and Call Waiting & Call Forward on Busy.

Keywords: Telephone Features, Feature Interactions, Formal Specifications, LOTOS.

1. Motivation and Background

The plain old telephone service (POTS) is used for establishing a communication session between two users. A telephone feature, such as *call waiting* (cw), *call forward on busy* (cfb), and *three way calling* (3wc), is defined as an added functionality of POTS. Augmenting POTS with a small set of features is considered to be a technically straightforward job. Both the behaviours of POTS and the features are analysed and decisions are taken as to how to integrate the features into POTS. Conflicts between any of the features are resolved on a case by case basis. However, as more and more features need to be integrated, as is the case for present and future telephone networks [Lata89], the task becomes more difficult. Features that perform their functions satisfactorily on their own are, in some instances, prevented from doing so in the presence of other features. This problem has become known as the *feature interaction problem* [BDCG89].

Investigations into the feature interaction problem fall into one of three complementary categories[CaVe93]: Detection, avoidance, and resolution. The objective of a detection approach is to analyse a set of independently specified features

---

1. This paper has also been accepted to the Second International Conference on Feature Interaction, Amsterdam, The Netherlands, May 1994.

and determine whether or not there are any conflicts between their joint behaviour [CaLi91], [Lee92], [BoLo93], [DaNa93]. An avoidance mechanism assumes that the causes of the interactions are known and an architectural or analytical approach is defined to prevent the manifestation of such interactions [MiTJ93]. The avoidance approach is most suitable in the early phases of specification and design of features. Finally, the objective of a resolution mechanism is to find appropriate solutions to interactions that manifest themselves at execution time [Cain92] [GrVe92].

This paper describes a method, based on the LOTOS specification language [BoBr87], [LoFH92], for detecting feature interactions at the specification level, in the context of single user single element features[CGLN93]. Central to our method are the concepts of *constraints* and *knowledge goals*. Constraints, which we have categorized as local, end-to-end, and global, are used to structure the specification so that new features can be added to the specification or removed from it with plausible ease [FaLS91]. While the concept of constraints is useful for structuring specifications, the concept of knowledge goals is useful for reasoning about telephone features, in order to detect feature interactions. It is based on the theories of *knowledge* which are being developed for understanding and reasoning about communication protocols and distributed systems [HaFa89], [HaMo90], [PaTa92]. In the knowledge-based approach, the evolution of the system can be described by the evolving knowledge of the components, about the state of other components, which collectively make up the global state. Thus, the state of knowledge in the system changes as a result of exchanging messages between the communicating components, and communication between the components depends on their knowledge about the system state. We use this concept to analyse combinations of features in order to detect the interactions between them.

In section 2, we describe our method for detecting interactions between independently specified features. In section 3, we demonstrate the application of our approach on two examples: *cw&cfb* and *cw&3wc*. We conclude with some thoughts regarding our research directions in section 4.

2. Using a LOTOS Environment to Detect Feature Interactions

Although the feature interaction problem has existed for quite many years, little attention was paid to it until it was explicitly defined [BDCG89]. Since then, a whole new field of interest is born. Due to the lack of space, we simply point the reader to some of the work of other researchers in this area. In particular, we mention the work of [HoSi88], [CaLi91], [Dwor91], [Cain92], [EKDB92], [GrVe92], [Inoue92], [Lee92], [DaNa93], and [Zave93]. Two other excellent sources of information are the special issues of IEEE Computer[Comp93] and IEEE Communications magazine [Magz93].

We begin the section by reviewing the constraint-oriented style that we had developed for specifying telephone systems in LOTOS. Then, we describe an

improvement to the structure of our specifications which makes it possible to easily integrate features into a telephone system, while still using the concept of constraints. We conclude the section by describing a reasoning mechanism which allows us to analyse the joint behaviour of features, for the purpose of detecting their interactions.

2.1 LOTOS Structure of the POTS Specification: An Overview

We have previously developed a LOTOS specification structure that is well suited for specifying the behaviour of telephone systems[FaLS91]. The structure is based on the constraint-oriented specification style [VSVB91], where we identified three types of constraints:

(1) *Local constraints* are used to enforce the appropriate sequences of events at each telephone, and are different according to whether the telephone is a *Caller* or a *Called*. Therefore local constraints are represented by the processes *Caller* and *Called* and an instance of each of these is associated with each telephone existing in the system. Because these two processes are independent of each other, they are composed by the interleaving operator |||.

(2) *End-to-End* constraints are related to each connection, and enforce the appropriate sequence of actions between telephones in a connection. For example, ringing at the *Called* must necessarily follow dialling at the *Caller*. Process *Controller* enforces these constraints. Because they must apply to both *Caller* and *Called*, we have the structure *(Caller ||| Called) || Controller*. Thus the controller must participate in every action of the *Caller*, as well as in every action of the *Called*, separately.

(3) *Global constraints* are system-wide constraints. In the POTS context, we identified one main constraint, which is the fact that at any time, a number is associated with at most one connection. Because global constraints, represented by a process *GlobalConstraints*, must be satisfied simultaneously over the whole system, we have the structure *Connections || GlobalConstraints*.

It should be stressed that the constraint-oriented style is purely a specification style, which allows to clearly separate the logical constraints a system must abide. It does not necessarily reflect an implementation architecture. To obtain an implementation architecture from a constraint-oriented structure, style transformations may be applied [VSVB91].

2.2 Enhancing the Structure of POTS Specifications

A graphical representation of the enhanced structure of POTS specifications is shown in Fig.1. The LOTOS specification of this structure, which handles only 4 users for the purposes of illustration, is shown in Fig. 2. This structure has the following characteristics: (1) Each user is represented by two processes, a caller side and a called side and each caller process is bound to its own controller. So, in the POTS case, each connection becomes: *(Caller(n) ||| Called) || Controller(n)*, where n is the subscriber number. (2) Contrary to the structure presented in [FaLS91], where the number of the

called user was passed to the called process at instantiation time, in this structure, the called process is now represented as a set of alternatives, where each alternative represents the called side of a user in the system. Clearly, since the controller offers to synchronize with only one called at a time, only a single alternative will offer the same value. As we will see in section 3.1, this structure is highly flexible for integrating new features into a telephone system. (3) the global constraints process participates in every action in which any instance of the controllers participates.

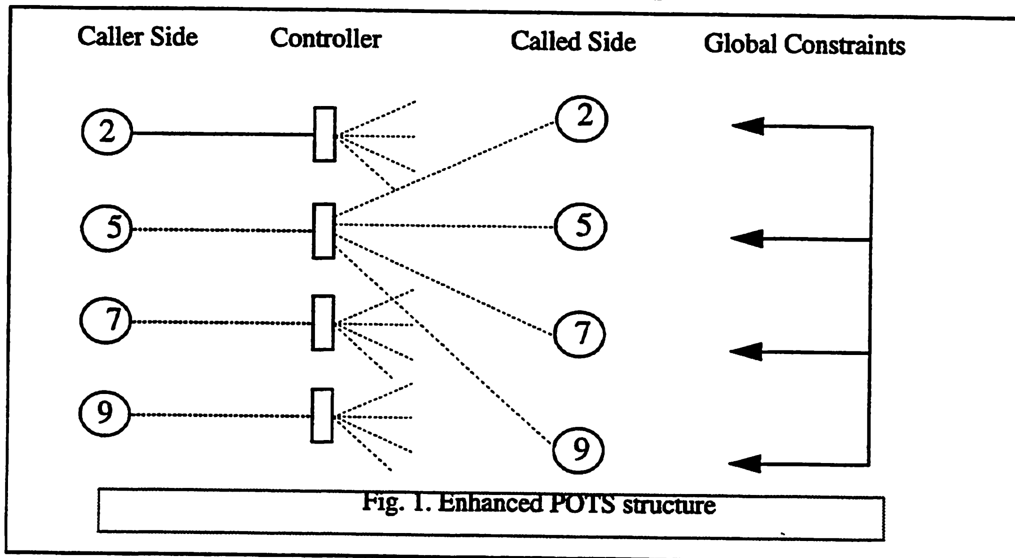


Fig. 1. Enhanced POTS structure

```
1    behaviour
2    (   ( Caller[Suser] (2) ||| Called [Suser](Users) )|| Controller [Suser] (2)
3        |||
4        ( Caller[Suser] (5) ||| Called [Suser](Users) ) || Controller [Suser] (5)
         |||
5        (Caller[Suser] (7) ||| Called [Suser](Users) ) || Controller [Suser] (7)
6        |||
7        ( Caller[Suser] (9) ||| Called [Suser](Users) ) || Controller [Suser] (9)     )
8    ||
9    GlobalConstraints[Suser](parameters)
10   where
11           process Caller[Suser](n: TelNo):noexit:= ...
12           process Called [Suser](Users: List): noexit:=
13               Called [Suser](2) [] Called [Suser](5) [] Called [Suser](7) [] Called [Suser] (9)
14           endproc (* Called *)
15           process Controller[Suser](n: Digit): noexit:= ...
16           process GlobalConstraints [Suser](Parameters: Sets)): noexit:= ...
```

Fig. 2. LOTOS specification of POTS using the enhanced structure.

2.2.1 Local Constraints

As mentioned, *Local constraints* are used to enforce the appropriate sequences of events within each process. For example, to specify a *Caller* process, the specifier

needs only to understand the events that are exchanged between the *Caller* process and its environment. At this stage, the specifier does not need to concentrate on *who* represents the environment or *which* processes will interact with the *Caller* process. These concerns are addressed at a later stage of the specification. By taking this view, we have in fact reinforced the concept of *separation of concerns*. In the case of specifications dealing with POTS [FaLS91] [BoLo93], *local constraints* were applied to the caller entity and the called entity only. Our experience has shown that the concept of local constraints can be used for specifying telephone features as well.

2.2.2 End-to-End Constraints

For a simple two-way call processing, the *end-to-end* constraints were used to synchronize the actions of two processes with respect to each other, most often the sender and the receiver. Our experience in writing the POTS specifications is that establishing a temporal order between two actions, one being offered by the caller and the other one by the called, is quite intuitive and simple. However, we recognize that expressing *end-to-end* constraints of the new structure may become more complicated, because there are more processes which may offer synchronization actions. And, it is still the specifier's task to impose a temporal order on a set of given actions. The specifier must then have some heuristics and guidelines at his/her disposal. A possible approach is to start by expressing the end-to-end constraints as *cause-effect*[Lin90] [NuPr93] rules.

2.2.3 Global Constraints

Finally, the *Global constraints* are at a higher level of abstraction than the end-to-end constraints, since they are imposed on the global behaviour of the system. In the simple two-way call processing model, the global constraints were restricted to enforcing *value* constraints between independent connections. In our new structure, global constraints gain an added importance. They enforce *control* constraints as well. Let us illustrate this with the following example, see Fig. 4. Suppose that user 2, who subscribes to *cw*, establishes a connection (A) with user 5. Also, suppose that, while 2 is talking to 5, 7 calls 2. Since 2 has *cw*, the global constraints process must manage the new connection (B), which consists of a new caller and shares the called side with the existing connection of 2. Therefore, the global constraint is responsible for switching *the control* between connection A and connection B, depending on what stage of the communication the users are in. For instance, when 7 dials 2 and 2 answers the call, the global constraint removes the communications between 5 and 2 from the set of active sessions and inserts it into the set of holding sessions. At the same time, it allows a communication session to be established between 7 and 2. When 2 and 7 finish their conversation, the global constraint reactivates the connection (A), between 2 and 5.

The structure that we have just defined exhibits the required flexibility. New features are defined in terms of their local constraints and their end-to-end constraints, while their global constraints are composed with the existing global constraints of the system. To specify a new feature, the specifier either instantiates actions that already

exist in the system, such as the *dial* action, or defines new actions, such as the *CwTone* action used in the definition of the *cw* feature. In the former case, the resulting global constraints on the *dial* action is the conjunction of the existing constraints and the new constraints. In the latter case, a new alternative action is added to the behaviour of the global constraint process.

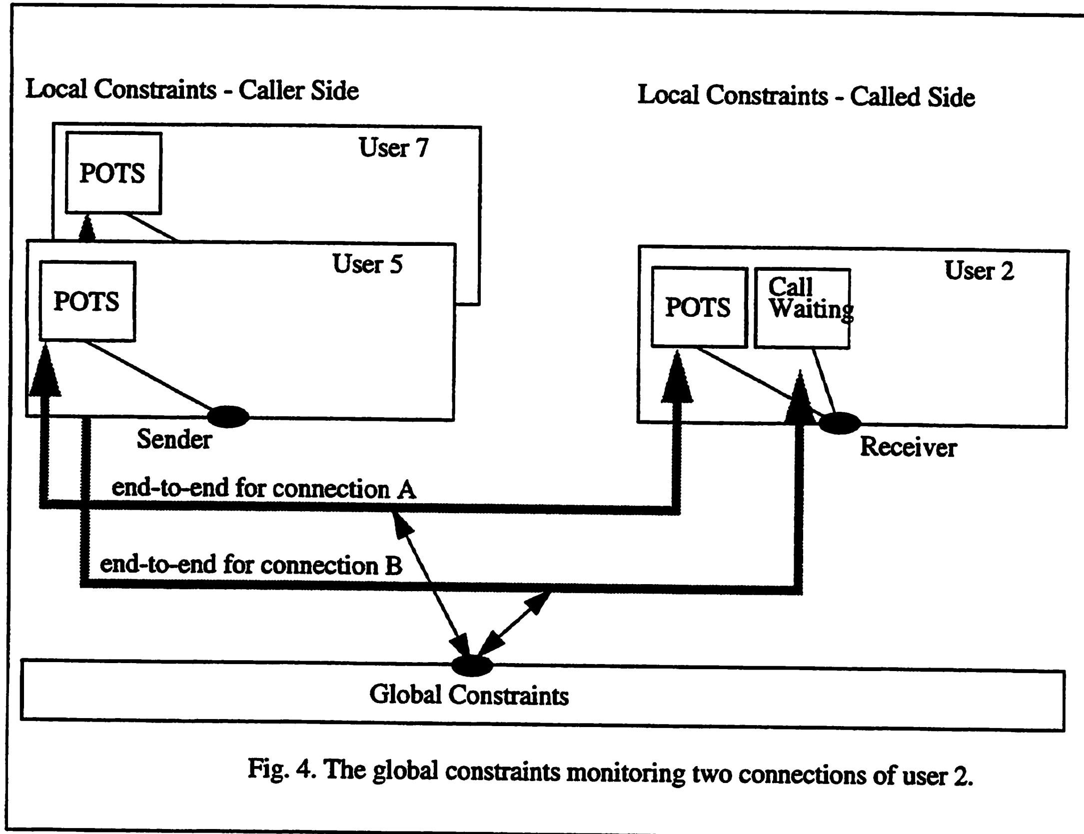


Fig. 4. The global constraints monitoring two connections of user 2.

2.3 Using Knowledge Goals to Reason about LOTOS Specifications

This section describes how to adapt the *knowledge-oriented* model of Halpern and Moses [HaMo90] and incorporate it into LOTOS specifications for telephony systems. The intuition we want to capture, by using the knowledge-based approach, is that the designer reasons about LOTOS processes in terms of how relevant information, from the local point of view (i.e., local constraints) of each process, becomes satisfied at certain points during the execution of the system. To analyze whether two features, say *cw* and *cfb*, interfere with each other, the designer defines a set of knowledge goals and verify their reachability, when both features are active. If any of the goals is unreachable, the designer concludes that a feature interaction (or design error) exists. Otherwise, no conclusion can be drawn from the analysis. In a way, this is similar to system testing. A test which fails to reveal an error does not indicate that the system under test is error free, it only means that the system is error free with respect to the assumption expressed by the test. Details of our analysis using two examples are given in section 3.

It is interesting to emphasize that each process reasons about the outside world only in terms of its local information. Therefore, a process moves from one state to another state based only on its knowledge. Similarly, it gains (or loses) new knowledge as it moves from one state to another. Also, notice that knowledge in this context is an

external notion, in the sense that processes do not acquire knowledge on their own nor are they able to analyze the knowledge state of other processes. For the purposes of analysis, the specifier is responsible for choosing the appropriate *knowledge goals* used to reason about the system.

3. Application of the Approach

In this section, we show how to apply our method to two examples of feature interactions: *cw* vs. *cfb* and *cw* vs. *3wc*. For each example, we show how to use the enhanced structure to integrate the two features into a single LOTOS specification, and then we show how to use the reasoning mechanism to detect their interactions. Only LOTOS segments which contribute significantly to the understanding of the integration and reasoning mechanisms are given.

3.1 Specifying Features in LOTOS

We have concluded from our experiments of specifying telephone features that most features act on behalf of either the caller side or the called side. From our structural and analytical points of view, some features which seem to act on behalf of both the caller side and the called side can be given only one of the two roles. An example of this is the *automatic recall* feature (not discussed in this paper). When a user is busy, this feature automatically returns the last incoming call when the subscriber's line becomes idle. We have classified this feature as having a caller role because its first action is to initiate a connection back to a user who has initiated a call.

3.1.1 Call Waiting

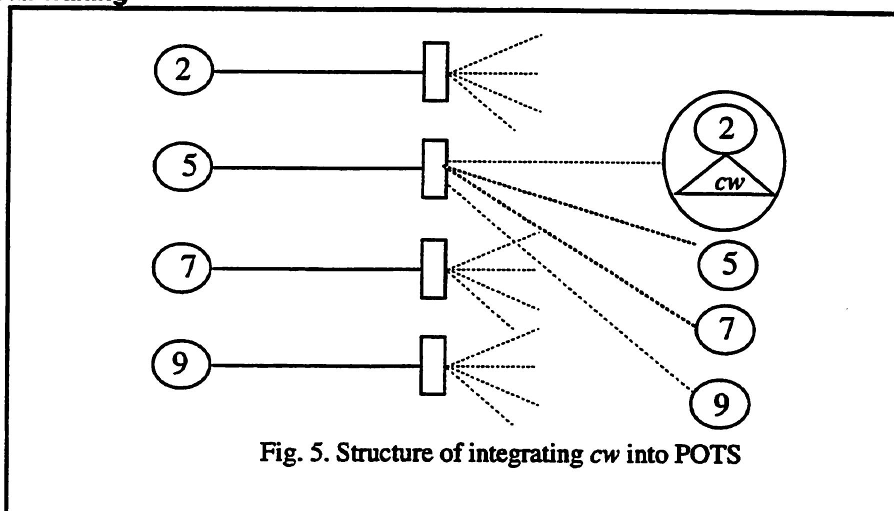


Fig. 5. Structure of integrating *cw* into POTS

Call Waiting is a feature which generates a call waiting tone, to alert a busy user that a second incoming call is waiting to be answered. The user may choose to answer the call, using a special *flashhook* signal, or may simply continue with the original communication and ignore the call waiting tone.

This feature has a called role. Therefore, first, we specify the behaviour of the feature in the context of POTS. Second, we modify the existing *Controller* so that it handles any new actions that the new feature participates in, such as *call waiting tone*. This results in a structure of the form: *(Caller(n)* ||| *Called)* || *ControllerCw(n)*. Third, we replace the existing definition of the controller with the new definition, which is now capable of handling POTS calls as well as subscribers with the call waiting feature. Finally, for each action in which the feature participates, we check that the predicates remain valid in the global constraints. Modifications of figure 1 are shown in Fig. 5 and result in the structure of Fig. 6.

```
1   behaviour
2   (   ( Caller[Suser] (2) ||| Called [Suser](Users) ) || Controller [Suser] (2)
3       |||
4       ( Caller[Suser] (5) ||| Called [Suser](Users) ) || Controller [Suser] (5)

5       |||
6       ( Caller[Suser] (7) ||| Called [Suser](Users) ) || Controller [Suser] (7)
7       |||
8       ( Caller[Suser] (9) ||| Called [Suser](Users) ) || Controller [Suser] (9)    )
9   ||
10  GlobalConstraints[Suser](parameters)
11  where
12      process Caller[Suser](n: TelNo):noexit:= ...
13      process Called [Suser](Users: List): noexit:=
14          CalledPots [Suser](2) [] CalledPots [Suser](5) [] CalledPots [Suser](7)
15          [] CalledPots [Suser] (9)
16          [] CalledCw(2)                          <-------------Added
17      endproc (* Called *)
18      process Controller[Suser](n: Digit): noexit:= ... <------------- Modified
19      process GlobalConstraints [Suser](Parameters: Sets)): noexit:= ... <— Modified
```

Fig. 6. LOTOS specification of *cw* within POTS.

3.1.2 Call Forward on Busy

Call Forward on Busy is a feature which allows a user, who is already involved in a conversation with a second user, to transfer his/her incoming calls to a predetermined third user. Depending on the specifier's intentions, the busy user may or may not be informed that a call transfer has occurred. This feature acts on behalf of the called side, so its specification is similar to that of call waiting. The modification of Fig. 1 results in a similar structure to that of call waiting, and is not shown.

3.1.3 Three Way Calling

Three way calling is a feature which allows a user, who is already involved in a conversation with a second user, to add a third user to the conversation. The subscriber of the feature must put the second user on hold, using a special *flashhook* signal, while establishing a communication with the third user. Once the communication is

established, a second *flashhook* brings the second user back to the conversation to form a 3 way communication.

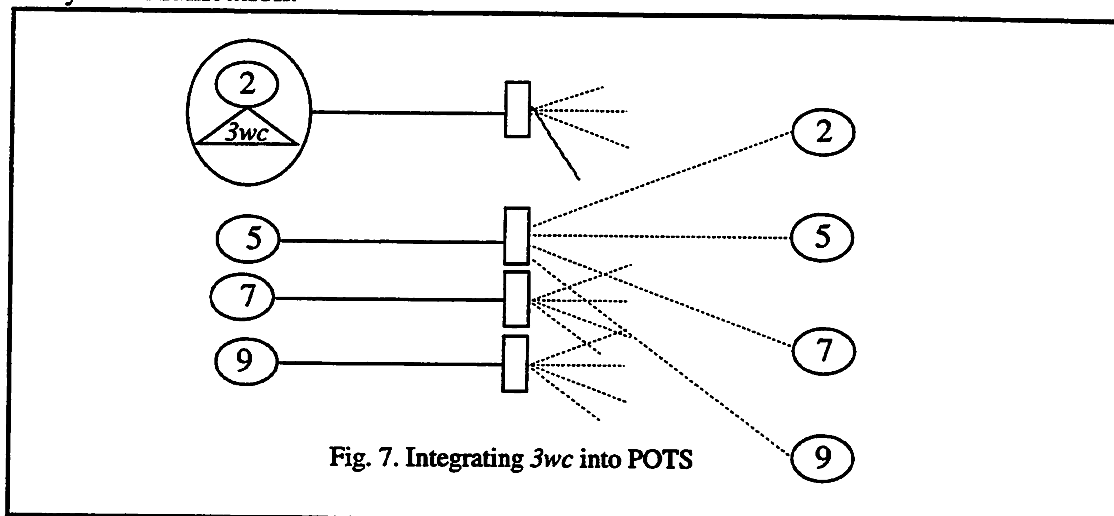


Fig. 7. Integrating *3wc* into POTS

This feature has a caller role. It is specified as follows. First, we specify the behaviour of the feature with respect to POTS. Second, we modify the existing *Controller* so that it handles any new actions that *3wc* participates in, such as *flashhook*. This results in a structure of the form: *(Twc(n)* ||| *Called)* || *Controller3wc(n)*, where *n* identifies the subscriber for which the feature is to be invoked. Third, we compose this new structure with the existing connections using the ||| operator. Finally, for each action in which the feature participates, we check that the predicates in the global constraints process remain valid. Extending our specification of Fig. 1, the structures of Figs. 7 and 8 result:

```
1    behaviour
2    (    ( Caller[Suser] (2) ||| Called [Suser](Users) ) || Controller [Suser] (2)
3         |||
4         (  Caller3wc[Suser] (2) ||| Called [Suser](Users) ) || Controller3wc [Suser](2)<-------- added
5         |||
6         ( Caller[Suser] (5) ||| Called [Suser](Users) ) || Controller [Suser] (5)
          |||
7         (  Caller[Suser] (7) ||| Called [Suser](Users) ) || Controller [Suser] (7)
8         |||
9         (  Caller[Suser] (9) ||| Called [Suser](Users) ) || Controller [Suser] (9)
10   )
11   ||
12   GlobalConstraints[Suser](parameters)
13   where
14            process Caller[Suser](n: TelNo):noexit:= ...
15            process Caller[Suser](n: TelNo):noexit:= ...
16            process Called [Suser](Users: List): noexit:=
17                 Called [Suser](2) [] Called [Suser](5) [] Called [Suser](7) [] Called [Suser] (9)
18            endproc (* Called *)
19            process Controller[Suser](n: Digit): noexit:= ...
20            process Controller3wc [Suser](n: Digit): noexit:= ... <-------- Added
21            process GlobalConstraints [Suser](Parameters: Sets)): noexit:= ...
```

Fig. 8. New LOTOS Structure of POTS

3.2 Analysing Features to Detect their Interactions

In this section, we present our method and show how it can be used to detect interactions between *cw&cfb* and *cw&3wc*. The method calls for the following steps to be carried out:

1• Specify each feature independently, within a POTS context;

2• Use the structure defined in section 2.2 to integrate both features into a single specification;

3• Define the knowledge goals to be reached in the reasoning phase; a knowledge goal is expressed as a LOTOS process which is composed in parallel with the specification obtained in 2 above.

4• Finally, simulate the system and check if the selected goals are reachable. A feature interaction (or design error) is detected if the selected goals are not reachable.

Assuming that points 1 and 2 above are completed successfully, let us proceed with point 3. In the two examples to follow, the designer has chosen to reason about the system in terms of subscribers' talking states. In other words, the designer knows that each feature, when active by itself within the context of POTS, can successfully reach its talking state, and the question she is attempting to answer is: can two features, if activated simultaneously by the same user, reach their talking states? as will be seen, the answer is no in both examples. Notice that if the designer fails to detect an interaction using a selected knowledge goal, it is strongly recommended that other goals be tried. So, it is the responsibility of the designer, using her design insights, to define the appropriate goals for reasoning about the system.

3.2.1 Call Waiting & Call Forward on Busy

Fig. 9. Integrating *cw* and *cfb*

Fig. 10 shows the modification of POTS by extending its functionality according to *cw* and *cfb*.Let us assume that only cw is active on 2. Then, if 7 calls 2 while 2 is talking to 5, one possible scenario is that 5 is put on hold and a talking session between 7 and 2 is established. This scenario is shown on the left side of Fig. 10. If we assume instead that only cfb is active, and using the same execution scenario, a talking session between 2 and 5 remains active and a talking session is established between 7 and 9. Therefore, our knowledge goal is defined as: Talk(2, 7) *and* Talk(7, 9). Since each of these two goals is reachable when only one feature is active, then they *must* also be reachable if cw and cfb are activated simultaneously with respect to their talking sessions.

To clarify this further, we denote by POTS+CW+CFB[gpots, gcw, gcfb] the extension of POTS by the features *cw* and *cfb,* where each feature is added to POTS separately; *gpots* is the gate through which all POTS events occur, *gcw* is the gate through which all *cw* events occur, *gcfb* is the gate through which all *cfb* events occur. Let the first part of the goal Talk[gpots, gcw](2, 7) be defined by the expression (gpots !7 !dials !2; gcw !2 !flashhook), as shown in the left branch of Fig. 10; let the second part of the goal Talk[gpots, gcfb](7, 9) be defined by the expression (gpots !7 !dials !2; gcfb

!9 !rings (from 7); gcfb !9 !answers), as shown in the right branch of Fig. 10. Therefore, if a deadlock (in the sense of LOTOS) occurs in the behaviour expression:
POTS+CW+CFB[gpots, gcw, gcfb] || (Talk[gpots, gcw] |[gpots]| Talk[gpots, gcfb]), then we can conclude that a feature interaction exists. The LOTOS specification and its execution tree are given as follows:

```
3   behaviour
4           gpots !7 !dials !2;
5           (  gcw !2 !flashhook; stop
6           []
7              gcfb !9 !rings; gcfb !9 !answers !7; stop
8           )
9           ||
10          (
11             gpots !7 !dials !2; gcw !2 !flashhook; stop
12             |[gpots]|
13             gpots !7 !dials !2; gcfb !9 !rings; gcfb !9 !answers !7; stop
14          )
Execution tree
1   gpots [4,11,13]
    | 1 gcw [5,11] DEADLOCK
    | 2 gcfb [7,13] DEADLOCK
```

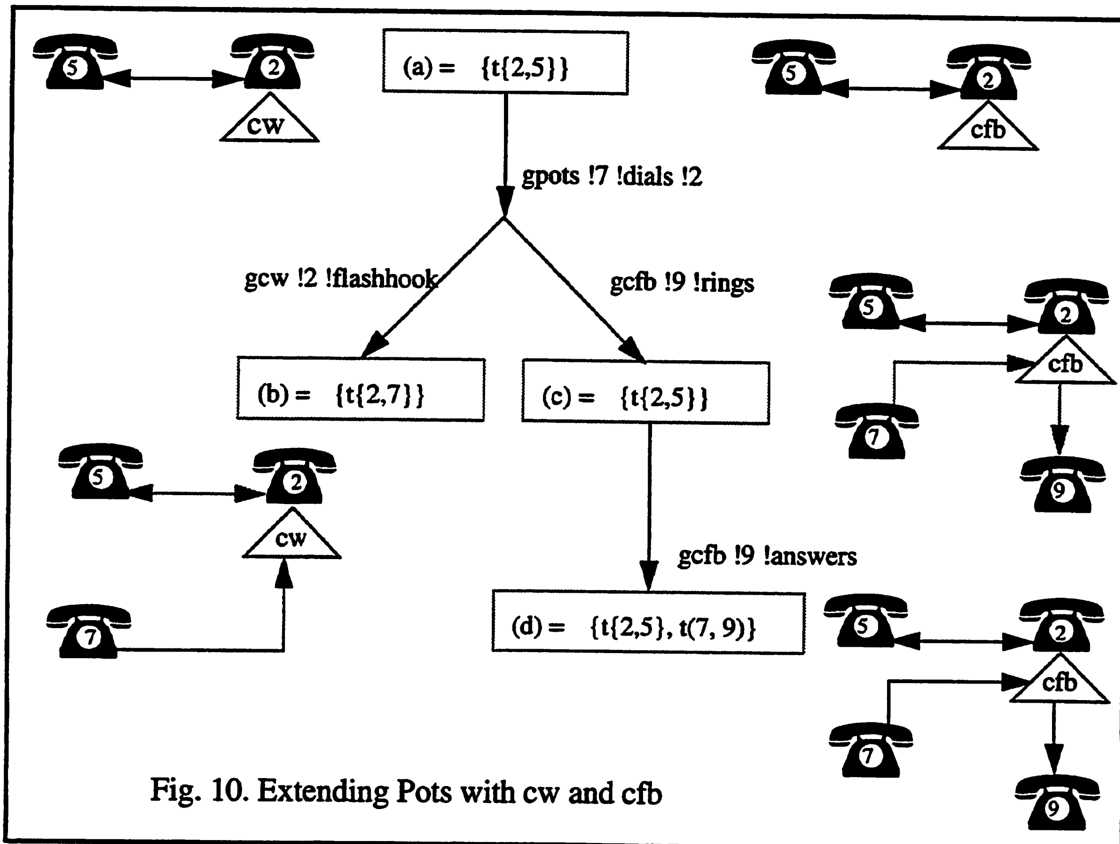


Fig. 10. Extending Pots with cw and cfb

3.2.2 Call Waiting & Three Way Calling

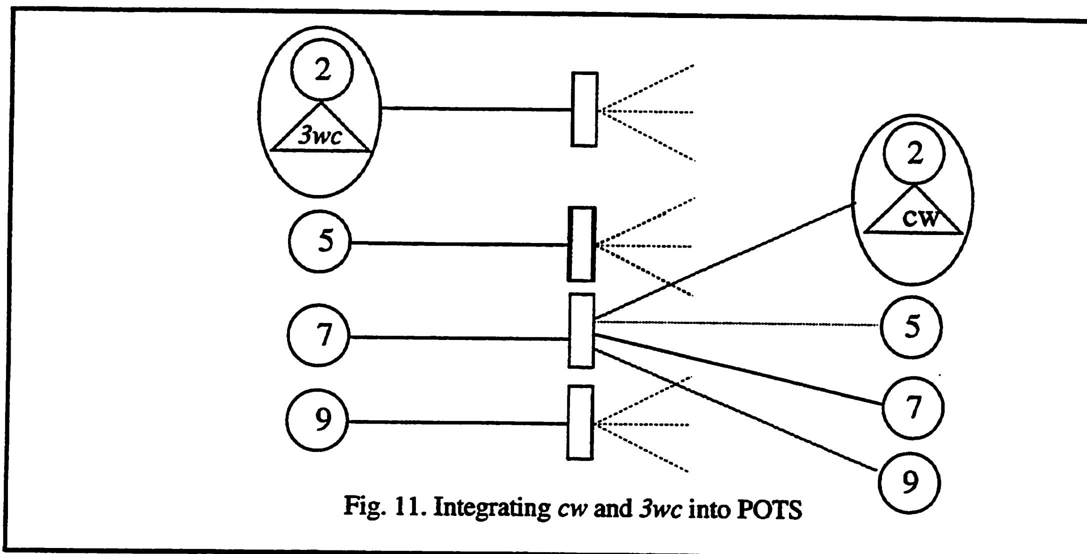


Fig. 11. Integrating *cw* and *3wc* into POTS

Our analysis for detecting the interaction between *cw* and *3wc* is similar to that of the previous example. Clearly, our objective is to know whether or not *cw* and *3wc* can be invoked simultaneously by 2 without causing a conflict between them.

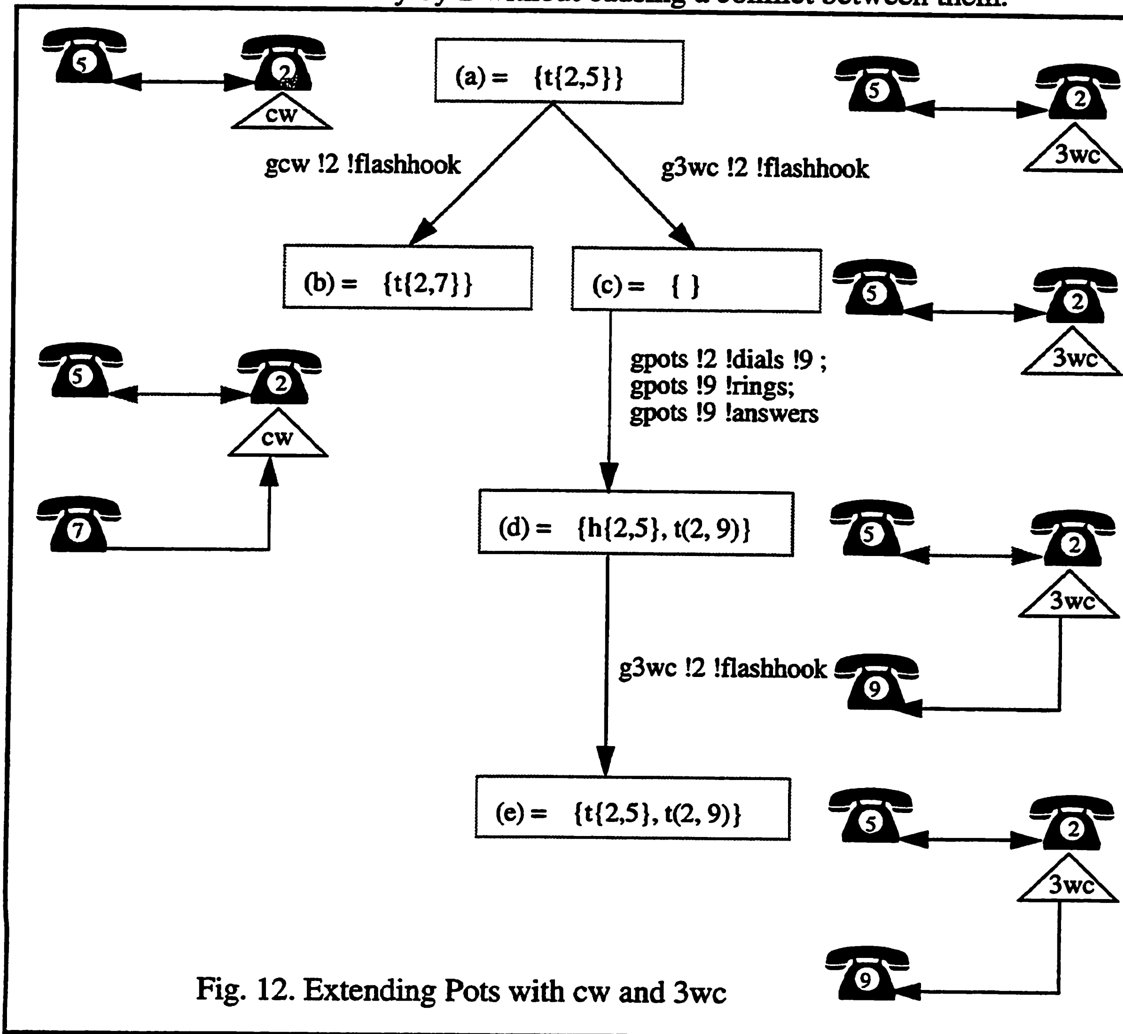


Fig. 12. Extending Pots with cw and 3wc

The left side of Fig. 12 is the same as the left side of Fig. 10. Concerning the right-hand side, assume that only 3wc is active, and that, using the same flashhook signal, 2 moves to a state where he may dial 9. To make the analysis easier, we assume that 9 is idle and answers the call. Therefore, a talking session between 2 and 9 is established. A second flashhook reestablishes the original talking session between 2 and 5. Therefore, our knowledge goal can be defined as: Talk(2, 7) *and* Talk(2, 9). As is in the previous example, we must show that the behaviour: POTS+CW+3WC[gpots, gcw, g3wc] || (Talk[gpots, gcw] |[gpots]| Talk[gpots, g3cw])) is deadlock free. In this case as well, a deadlock is reached as can be easily verified by executing the above behaviour expression. We conclude that a feature interaction exists.

4. Conclusions and Research Directions

We have proposed a method, based on a formal approach, for detecting feature interactions at the specification level, for single user single element features. Structurally, the approach uses three types of constraints: local constraints, end-to-end constraints, and global constraints. This structure allows the integration of new feature specifications into existing ones simply by classifying their roles as *caller* or *called* and expressing their global constraints as a conjunction. This structure is possible because of LOTOS's multiway synchronization mechanism, which also offers the flexibility to describe a system as a composition of constraints. Analytically, the approach is based on a reasoning mechanism which allows the specifier to analyse features, for the purpose of detecting their interactions, based on knowledge goals, where each goal is expressed as a LOTOS process. Our interpretation, in this context, is that conflicts between features correspond to deadlock situations in the LOTOS sense.

Important items for future research are adapting our approach to more realistic examples, extending the technique to other types of feature interactions, and making the technique more automatic. i.e., less dependent on designer's insight regarding where the problem might be found.

Acknowledgment. Funding sources for our work include the Natural Sciences and Engineering Research Council of Canada, the Telecommunications Research Institute of Ontario, Bellcore, Bell-Northern Research, and the Canadian Department of Communications. We like to acknowledge the many fruitful discussions that we had with Bernard Stepien and members of our LOTOS group. Also, comments from the referees have led to improvements of the content of this paper.

5. References

[BDCG89] T.F. Bowen, F.S. Dworak, C.H. Chow, N. Griffeth, G.E. Herman, and Y-J. Lin, The Feature Interaction Problem in Telecommunications Systems, 7th International Conference on Software Engineering for Telecommunication

Switching Systems, July 1989, 59-62.

[BoBr87] Bolognesi, B.,Brinksma, E. Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14, 1987, 25-59.

[BoLo93] R. Boumezbeur, L. Logrippo, Specifying Telephone Systems in LOTOS, IEEE Communications Magazine, Aug. 1993, 38-45. E. J. Cameron, N. Griffeth, Y. Lin, M. E. Nilson, W. K. Schnure, H. Velthuijsen, A Feature Interaction Benchmark for IN and Beyond, IEEE Communications, vol. 31, No. 3, 64-69, March 1993.

[CaLi91] E. J. Cameron and Y.J. Lin, A Real-Time Transition Model for Analyzing Behavioral Compatibility of Telecommunications Services. In Proceedings of the ACM SIGSOFT 1991 Conference on Software for Critical Systems, pp. 101-111, December 1991, New Orleans, Louisiana.

[CaVe93] J. Cameron and H. Velthuijsen, Feature Interactions in Telecommunications Systems, IEEE Communications Magazine, Aug. 1993, 18-23.

[Cain92] M. Cain, Managing Run-Time Interactions Between Call-Processing Features, IEEE Communications Magazine, pp. 44-50, February 1992.

[Comp93] IEEE Computer, Special Issue on Feature Interactions in Telecomunications Systems, Aug. 1993.

[DaNa93] O. Dahl and E. Najm, Specification and Detection of IN Service Interference Using LOTOS, to appear in the proceedings of Forte '93, Boston.

[Dwor91] F. S. Dworak, Approaches to Detecting and Resolving Feature Interactions, GLOBECOM 1991, pp. 1371-1377.

[EKDB92] M. Erradi, F. Khendek, R. Dsouli, and G. V. Bochmann, Dynamic Extension of Object-Oriented Distributed System Specifications, First International Workshop on Feature Interactions in Telecommunications Software Systems, Florida, 1992, 116-132.

[FaLS91] Faci, L. Logrippo and B. Stepien,Formal Specifications of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach, Computer Networks and ISDN Systems, 21, 52-67, North Holland, 1991.

[GrVe92] N. D. Griffeth and H. Velthuijsen, The negotiating agent model for Rapid Feature Development, Proceedings of the 8th International Conference on Software Engineering for Telecommunications Systems and Services, Florence, Italy, March/April 1992.

[HaFa89] J. Y. Halpern and R. Fagin, Modelling Knowledge and action in distributed systems, Distributed Computing, 3, 159-177, 1989.

[HaMo90] J. Y. Halpern and Y. Moses, Knowledge and Comon Knowledge in a Distributed Environment, JACM, Vol. 37, No. 3, 549-587, July 1990.

[HoSi88] S. Homayoon and H. Singh, Methods of Addressing the Interactions of Intelligent Network Services With Embedded Switch Services, IEEE Communications Magazine, pp. 42-47, Dec. 1988.

[Inoue92] Y. Inoue, K. Takami, and T. Ohta, Method for Supporting Detection and Elimination of Feature Interaction in a Telecommunication System, First International Workshop on Feature Interactions in Telecommunications

Software Systems, Florida, 1992, 61-81.

[Lata89] LATA Switching Systems Generic Requirements (LSSGR), Bellcore, TR-TSY-000064, FSD 00-00-0100, July 1989.

[Lee92] A. Lee, Formal Specification and Analysis of Intelligent Network Services and their Interaction, Ph. D. Thesis, Dept. of Computer Science, University of Queensland, 1993.

[Lin90] Y. J. Lin, Analyzing Service Specifications Based upon the Logic Programming Paradigm, In Proceedings of the IEEE GLOBECOM 1990, pp. 651-655, December 1990, San Diego, California.

[LoFH92] L. Logrippo, M. Faci and M. Haj-Hussein, An Introduction to LOTOS: Learning by Examples, Computer Networks & ISDN Systems, Vol. 23, No. 5, 1992, pp. 325-342.

[Magz93] IEEE Communications Magazine, Special Issue on Feature Interactions in Telecomunications Systems, Aug. 1993.

[MiTJ93] J. Mierop, S. Tax, R. Janmaat, Service Interaction in an Object Oriented Environment, IEEE Communications Magazine, Aug. 1993.

[NuPr93] K. Nursimulu and R. L. Probert, Cause-Effect Validation of Telecommunications Service Requirements, Technical Report TR-93-15, University of Ottawa, Dept. of Computer Science, October 1993.

[PaTa92] P. Panangaden and K. Taylor, Concurrent Common Knowledge: Defining Agreement for Asynchronous Systems, Distributed Computing, 6, 73-93, 1992.

[VSVB91] C.A. Vissers, G. Scollo, M. van Sinderen, E. Brinksma, Specification Styles in Distributed Systems Design and Verification, Theoretical Computer Science 89, 1991, 179-206.

[Zave93] P. Zave, Feature Interactions and Formal Specifications in Telecommunications, IEEE Computer Magazine, Aug. 1993, 18-23.

"Plain Old Telephone System Service" (POTS) : Validation avec MEC[1]

Srećko Brlek, Richard Mallette
LaCIM, Département de Mathématiques et Informatique, UQAM,
Montréal, P.Q. H3C 3P8, CANADA

11 février 1994

Résumé
Le but de ce papier est de montrer l'utilisation de MEC, dans la validation des systèmes de transitions associés à la description LOTOS du modèle POTS.

1 Introduction

MEC est un outil d'analyse sur les systèmes de transitions [Ar89]. Il permet de vérifier des propriétés et en particulier s'il existe des chemins pouvant diriger le système vers un état « deadlock ». LOTOS permet de spécifier des processus ayant des contraintes de temps ou d'événements. Nous avons l'intention de montrer que MEC est un outil intéressant afin de valider les spécifications LOTOS pour un système téléphonique élémentaire[Ar90]. Dans un premier temps nous allons traduire les spécifications LOTOS en systèmes de transitions, ensuite transcrire ces systèmes dans un langage reconnu par MEC. Finalement on explicitera les « deadlock ». Nous allons appliquer la méthode à la spécification du POTS telle que donnée par Loggripo [FLS91].

2 De LOTOS aux systèmes de transitions

LOTOS (Language of Temporal Ordering Specification) s'apparente de près à CCS. LOTOS fourni un ensemble de règles pouvant décrire des processus réagissant et s'échangeant des

[1] Recherches financés par l'Action Concertée sur les Méthodes Mathématiques pour la synthèse des systèmes informatiques., FCAR-BNR-CRSNG.

Nous utilisons les chiffres 1 à 5 pour identifier les étapes du processus LOTOS: 1- entrée du processus, 2,3,4 sont les trois choix possibles et 5 est l'action qui peut interrompe en tout temps le choix fait entre 2,3 et 4. Dans l'arbre de décision nous retrouvons les étapes aux sommets du graphe, finalement l'automate reflète l'arbre d'actions.

3. Le premier modèle de POTS

Le premier modèle de Luigi Logrippo [FLS91] utilise un processus contrôleur associé à un processus manipulant une liste d'utilisateurs. Les utilisateurs communiquent via les processus "Caller" et "Called". Le schéma 1 présente les relations entre ces processus.

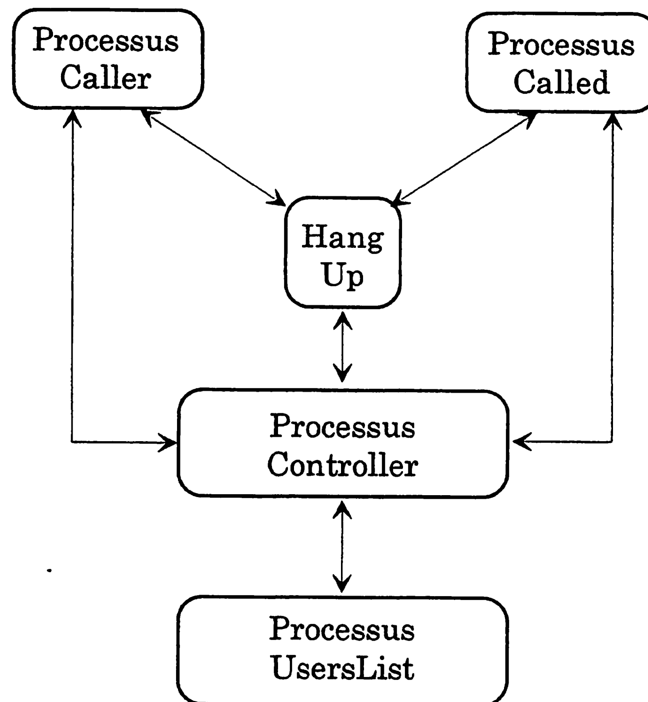


Schéma 1 : relations entre les processus

Le schéma 2 présente le système de transitions associé au processus contrôleur du schéma 1. Le schéma 3 celui de processus "UserListHandler" et le schéma 4 celui du processus appelant. Le processus appelé est représenté sur le schéma 5.

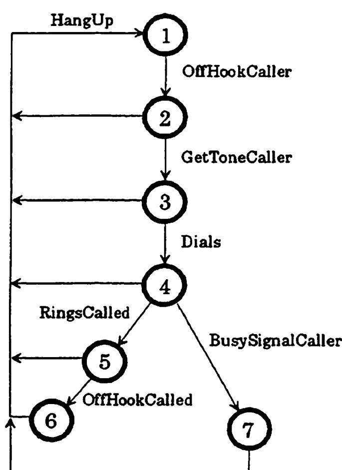


Schéma 2: Controller

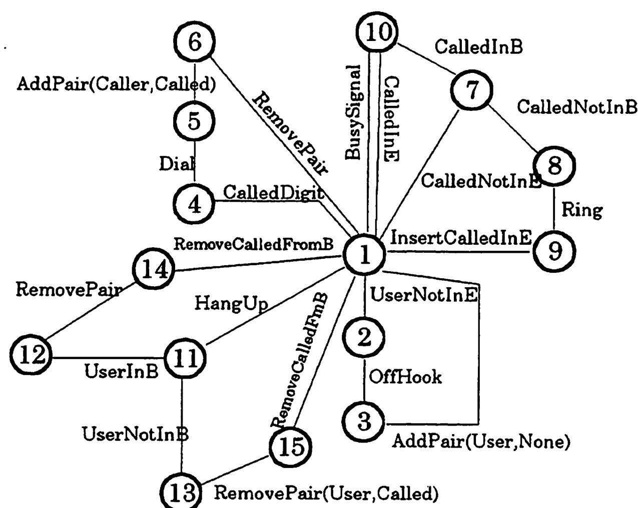


Schéma 3: ListUsersHandler

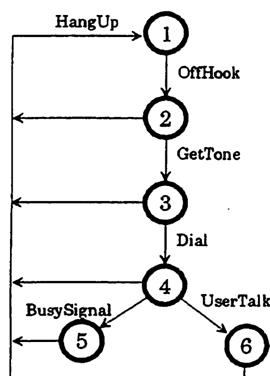


Schéma 4: Caller

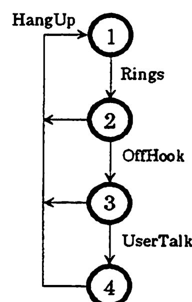


Schéma 5: Called

Dans ce système, nous voyons immédiatement qu'il faut synchroniser certaines actions. Exemple, lorsqu'un appel est terminé ou coupé, il faut que tous les processus se retrouve dans l'état initial de départ, donc le « HangUp » est synchronisé sur tout les processus. Il en est de même pour le fait de décrocher le combiné, le contrôleur et le « Caller » sont en synchronisation, (OffHookCaller, OffHook), pour produire les événements nécessaires à une reconnaissance du début d'appel. Dans cette suite d'événement, la tonalité nécessite une synchronisation entre le contrôleur et l'appelant, (GetToneCaller, GetTone), Lorsque L'appelant fera la composition numérique, la synchronisation se fera par (Dials, Dial), entre le contrôleur et l'appelant. Lors de la vérification dans la liste pour un appelé libre ou occupé, une synchronisation est nécessaire sur les signaux (BusySignalCaller, BusySignal). Si l'appelé est libre, alors le contrôleur envoie un signal de sonnerie, ceci nécessite une synchronisation entre les signaux (RingsCalled, Rings). Si

l'appelant décroche le combiné pour répondre, alors le contrôleur doit agir et pour ce faire il y a synchronisation sur les signaux (OffHookCalled, OffHook). Finalement la synchronisation des signaux (Talk, Talk) démontre l'action d'une communication en cours.

3.1 Modèle MEC

Les systèmes de transitions des schémas 2, 3, 4, et 5 sont traduits dans la syntaxe MEC comme suit:

```
transition_system HangUp;
1|-   e->1,Hangup->1;
<initial={1}>.

transition_system UsersList;
1|-e->1,getTone->1,Talk->1,UserNotInE->2,CalledDigit->4,CalledNotInE->7,CalledInE1->10,Hangup->11;
2|-e->2,OffHook->3;
3|-e->3,AddUserNone->1;
4|-e->4,Dial->5;
5|-e->5,AddCallerCalled->6;
6|-e->6,RemoveUserN1->1;
7|-e->7,CalledNotInB->8,CalledInE->10;
8|-e->8,Ring->9;
9|-e->9,InsertCalledInE->1;
10|-e->10,BusySignal->1;
11|-e->11,UserInB->12,UserNotInB->13;
12|-e->12,RemoveUserN2->14;
13|-e->13,RemoveUserCald->15;
14|-e->14,RemoveCaldF1->1;
15|-e->15,RemoveCaldF2->1;
<initial={1}>.

transition_system Controller;
1|-e->1,Hangup->1,OffHookA->2;
2|-e->2,Hangup->1,GetToneA->3;
3|-e->3,Hangup->1,Dials->4;
4|-e->4,Hangup->1,RingsZ->5,BusySignal->7;
5|-e->5,Hangup->1,OffHookZ->6;
6|-e->6,Hangup->1,Disable->1;
7|-e->7,Hangup->1,Disable->1;
<initial={1}>.

transition_system Caller;
\*Processus de l'appelant*\
1|-e->1,Hangup->1,OffHook->2;
2|-e->2,Hangup->1,GetTone->3;
3|-e->3,Hangup->1,Dial->4;
4|-e->4,Hangup->1,BusySignal->5,UsersTalk->6;
5|-e->5,Hangup->1;
6|-e->6,Hangup->1,Talk->6;
<initial={1}>.
```

```
transition_system Called;
\*Processus de l appele *\
1|-e->1,Hangup->1,Ring->2;
2|-e->2,Hangup->1,Offhook->3;
3|-e->3,Hangup->1,UsersTalk->4;
4|-e->4,Hangup->1,Talk->4;
<initial={1}>.

transition_system tr;

1|-e->1,Hangup->1,OffHookA->1,TalkUL->1,GetToneA->1,DialA->1,BusySignalUL->1,RingUL->1,
  OffHookZ->1,TalkUL->1,UserNotInE->1,AddUserNone->1,InsertCalledInE->1,CalledNotInE->1,
  CalledInE->1,CalledNotInB->1,UserNotInB->1,UserInB ->1,RemoveCaldF1->1,
  RemoveCaldF2->1,CalledDigit->1,CalledInE1->1,RemoveUserCald->1,RemoveUserN1->1,
  RemoveUserN2->1,AddCallerCalled->1,Disable1->1,Disable2 ->1;
<initial={1}>.

synchronization_system vecteurs
<width=6;list=
(HangUp,  UsersList         ,Controller  ,Caller      ,Called    ,tr  )>;
(Hangup   .Hangup           .Hangup      .Hangup      .Hangup    .Hangup );
(e      .OffHook            .OffHookA    .OffHook     .e         .OffHookA);
(e        .GetTone          .GetToneA    .GetTone     .e         .GetToneA);
(e        .Dial             .Dials       .Dial        .e         .DialA);
(e        .BusySignal       .BusySignal  .BusySignal  .e         .BusySignalUL);
(e        .Ring             .RingsZ      .e           .Ring      .RingUL);
(e        .e                .OffHookZ    .e           .OffHook   .OffHookZ);
(e        .Talk             .e           .Talk        .Talk      .TalkUL);
(e        .UserNotInE       .e           .e           .e         .UserNotInE);
(e        .AddUserNone      .e           .e           .e         .AddUserNone);
(e        .InsertCalledInE  .e           .e           .e         .InsertCalledInE);
(e        .CalledNotInE     .e           .e           .e         .CalledNotInE);
(e        .CalledInE        .e           .e           .e         .CalledInE);
(e        .CalledNotInB     .e           .e           .e         .CalledNotInB);
(e        .UserNotInB       .e           .e           .e         .UserNotInB);
(e        .UserInB          .e           .e           .e         .UserInB);
(e        .RemoveCaldF1     .e           .e           .e         .RemoveCaldF1);
(e        .RemoveCaldF2     .e           .e           .e         .RemoveCaldF2  );
(e        .CalledDigit      .e           .e           .e         .CalledDigit);
(e        .RemoveUserCald   .e           .e           .e         .RemoveUserCald);
(e        .RemoveUserN1     .e           .e           .e         .RemoveUserN1);
(e        .RemoveUserN2     .e           .e           .e         .RemoveUserN2);
(e        .AddCallerCalled  .e           .e           .e         .AddCallerCalled);
(e        .CalledInE1       .e           .e           .e         .CalledInE1);
(e        .e                .e           .Disable     .e         .Disable1);
(e        .e                .e           .e           .Disable   .Disable2).
```

3.2 Validation

Les commandes suivantes permettent de vérifier s'il existe des « deadlock » ou si tous les états sont utlisisés.

```
sync(vecteurs,zp);
dts(zp);
dead:=*-src(*);
proj(6,exist,rsrc(initial)\/*);
dts(tr);
notex:=*-exist;
ar_ts(err2,*,notex);
t1:=trace(dead,*);
dead1:=tgt(t1);
probleme2:=dead-dead1;
t2:=trace(probleme2,*);
dead2:=tgt(t2);
probleme3:=probleme2-dead2;
t3:=trace(probleme3,*);
```

La commande « dts(zp);» affiche un produit synchronisé de 8 états et de 22 transitions, ce qui semble anormal. En effet la commande «dead:=*-src(*); » indique qu'il y a sept « deadlock », ils sont tout générés par le processus "ListUsersHandler".

Les états bloquants sont : (1.10.1.1.1.1), (1.4.1.1.1.1), (1.8.1.1.1.1), (1.10.2.2.1.1), (1.2.2.2.1.1), (1.4.2.2.1.1), (1.8.2.2.1.1), ce sont les états précédants les transitions Ring, BusySignal et Dial du processus ListUsersHandler qui bloque le système alors que le contrôleur et l'appelant sont soit dans l'état 1 ou immédiatement après un « OffHook ». Si nous faisons abstraction de la gestion de la liste dans le ListUsersHandler, alors il n'y a plus d'états bloquants. La trop grande restriction qu'a le « ListUserHandler » fait en sorte que le système peut se retrouver dans la situation d'avoir vérifié si un appelé est dans la liste avant même qu'il y ait possibilité de synchronisation avec la transition « dial » du contrôleur donc n'ayant pas encore reçu l'identification de l'appelé, il en est de même pour la transition Dial et Ring. Cette restriction est que pour générer un Offhook, il faut que le « ListUserHandler » ait rencontré certaine condition, c'est-à-dire que l'utilisateur ne soit pas dans la liste, Alors toutes autres actions exécutées avant un Offhook peut amener le système dans un état bloquant du « ListUserHandler ».

Si nous enlevons le processus ListUsersHandler, le système étant simple n'a pas de « deadlock ». Par contre c'est cette liste qui gère les activités des usagers. Par cette liste, le contrôleur sait si un usager est occupé ou libre. Il est difficile de faire abstraction de la liste dans le modèle sans diminuer l'information pertinente.

D'ailleurs dans le deuxième modèle de POTS, les auteurs, ont pris en considération, le problème que représente une liste et ils ont plutôt, introduit dans le processus station , l'état d'être libre ou occupé.

4 Le deuxième modèle de POTS

Le deuxième modèle est tiré de [StL93]. Ce modèle n'utilise pas de liste comme dans le premier modèle. Dans la traduction vers les systèmes de transitions, nous avons généralisé le concept de terminaison d'appel. Nous avons utilisé un niveau d'abstraction qui permet de ne pas détailler les quatre façons de terminer une liaison ou de terminer une demande de liaison.

Dans ce modèle, sous LOTOS, il y a un processus contrôleur qui peut être appelé plusieurs fois, nous retrouvons une forme de récursivité qui permet plusieurs instantiations du processus. Pour refléter ce fait nous avons dans un premier temps modélisé un système de transitions n'ayant qu'un seul contrôleur et deux stations. Dans un deuxième temps nous avons modélisé avec deux contrôleurs et deux stations. Dans les deux cas nous ne rencontrons aucun "deadlock". Voyons les processus écrit en LOTOS tel que présentés dans [StL93].

4.1 Modèle LOTOS:

```
process
station[b,g,n,t,tn](N:phone_number
):noexit:=
                                   (
call_initiator_station[b,g,n,t,tn](N)
  []
call_responder_station[b,g,n,t,tn](N
)
 )
 >> station[b,g,n,t,tn](N)
endproc

process
call_initiator_station[b,g,n,t,tn]

(N:phone_number):noexit :=
 g!N!off_hook;
 (
  (
   (g!N!tone;
    g!N!dial?C:phone_number;
    n!N!conreq!C;

n!N!connect;
    talking[g](N)
   )
   |||
   continuous_busy_signal[b](N)
  )
[>station_call_termination[t,tn](N)
 )
endproc
process continuous_busy_signal[b]

(N:phone_number):noexit:=
 b!N!busy;
 continuous_busy_signal[b](N)
endproc


process
talking[g](N:phone_number):noexit
:=
g!n!voice;talking[g](N)
endproc

process
call_responder_station[b,g,n,t,tn]
(C:phone_number):noexit:=
 n?N:phone_number!ring!C;
 (
  (
   ( g!C!answer;
     n!C!connect;
     talking[g](C)
   )
   |||
   continuous_busy_signal[b](C)
  )
[>station_call_termination[t,tn](C)
 )
endproc
process controller[b,n,tn]:noexit:=
n?N:phone_number!conreq?C:phon
e_number;
 connection_control[b,n,tn]
endproc


process   connection_control[b,n,tn]
(N,C:phone_number):noexit:=
 (
  (ring_a_free_number[n](N,C)
   []
   detect_busy_signal[b,n,tn](N,C)
   []
   detect_not_in_service[b,tn](N,C)
  )
  [>control_termination[tn]
 )
 |||
 controller[b,t,tn]
endproc


process ring_a_free_number[n]

(N,C:phone_number):noexit:=
 n!N!ring!C;
```

```
 n!C!connect;
 n!N!connect;
 stop
endproc


process detect_busy_signal[b,tn]

(N,C:phone_number):noexit:=
 b!C!busy;
 ( tn!N!disind?S:status;stop
  []
  tn!N!disreq?S:status;stop
 )
endproc


process detect_not_in_service[b,tn]

(N,C:phone_number):noexit:=
 b!C!not_in_service;
 ( tn!N!disind?S:status;stop
  []
  tn!N!disreq?S:status;stop
 )
```

4.2 Système de transitions

Le schéma 6 montre les relations entre les processus pour ce deuxième modèle POTS.

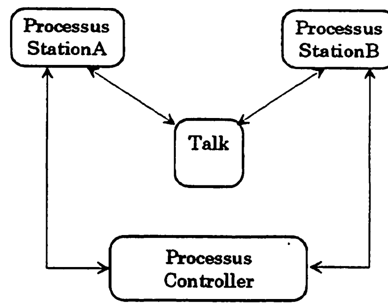


Schéma 6: relations entre les processus.

Les schémas 7 et 8 montrent les systèmes de transitions utilisés pour la modélisation sous MEC. Lors de la deuxième approche, avec deux contrôleurs et deux stations, nous dupliquons les systèmes de transitions des schémas.

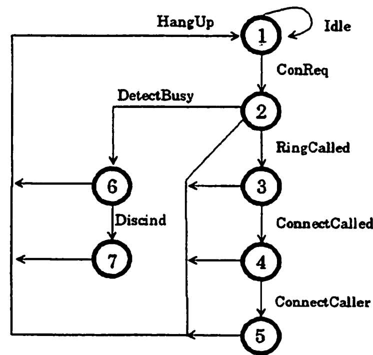


Schéma 7 : Contrôleur

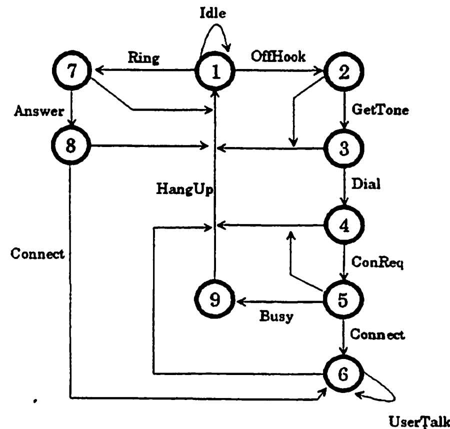


Schéma 8: Station

4.3 Modèle MEC

Voici le modèle ayant un contrôleur et deux stations:

```
transition_system UserA;
1|-  e->1,
        OffHook->2,
        Answer->3;
2|-     e->2,
        Hangup->1,
        Dial->3;
3|-     e->3,
        Hangup->1,
        UserTalk->3;
<initial={1}>.
```

```
transition_system UserB;
1|-  e->1,
        OffHook->2,
        Answer->3;
2|-     e->2,
        Hangup->1,
        Dial->3;
3|-     e->3,
        Hangup->1,
        UserTalk->3;
<initial={1}>.
```

```
transition_system Controller;

1|-     e->1,
        ConReq->2;
2|-     e->2,
        CHangup->1,
        RingCalled->3,
        DetectBusy->6;
3|-     e->3,
        CHangup->1,
        ConnectZ->4;
4|-     e->4,
        CHangup->1,
        ConnectA->5;
5|      e->5,
        CHangup->1;
6|-     e->6,
        CHangup->1,
        Discind->7;
7|-     e->7,
        CHangup->1;
<initial={1}>.
```

```
transition_system StationA;

1|-     e->1,
        OffHook->2,
        Ring->7;
2|-     e->2,
        Hangup->1,
        GetTone->3;
3|-     e->3,
        Hangup->1,
        Dial->4;
4|-     e->4,
        Hangup->1,
        ConReq->5;
5|-     e->5,
        Hangup->1,
        ConnectA->6,
        Busy->9;
6|-     e->6,
        Hangup->1,
        Talk->6;
7|-     e->7,
        Hangup->1,
        Answer->8;
8|-     e->8,
        Hangup->1,
        ConnectZ->6;
9|-     e->9,
        Hangup->1;
<initial={1}>.
```

```
transition_system StationB;

1|-     e->1,
        OffHook->2,
        Ring->7;
2|-     e->2,
        Hangup->1,
        GetTone->3;
3|-     e->3,
        Hangup->1,
        Dial->4;
4|-     e->4,
        Hangup->1,
        ConReq->5;
5|-     e->5,
```

```
        Hangup->1,
        ConnectA->6,
        Busy->9;
6|-     e->6,
        Hangup->1,
        Talk->6;
7|-     e->7,
        Hangup->1,
        Answer->8;
8|-     e->8,
        Hangup->1,
        ConnectZ->6;
9|-     e->9,
        Hangup->1;
<initial={1}>.
```

```
transition_system tr;

1|-e->1,Hangup->1,OffHookA->1,
Talk->1,GetToneA->1,DialA->1,BusyA->1,
RingA->1,AnswerA->1,
ConReq->1,DetectB->1,
ConnectZb->1,ConnectAb->1,
CHangup->1,OffHookB->1,
DialB->1,AnswerB->1,
ConReqB->1,GetToneB->1,
BusyB->1,RingB->1,
ConnectZa->1,  ConnectAa->1;
<initial={1}>.
```

```
synchronization_system vecteurs
<width=6;list=

(UserA,UserB,Controller,StationA,StationB ,tr)>;
(OffHook.e.e.OffHook.e.OffHookA);
(e.e.e.GetTone.e.GetToneA);
(Dial.e.e.Dial.e.DialA);
(e.e.ConReq .ConReq.e.ConReqA);
(e.e.ConnectA.ConnectA.e.ConnectAa);
(Answer.e.e .Answer.e.AnswerA);
(e.e.ConnectZ.ConnectZ.e.ConnectZa);
(e.e.RingCalled.Ring.e.RingA);
(e.e.Discind.Busy.e.BusyA);
(e.OffHook.e.e.OffHook.OffHookB);
(e.e.e.e.GetTone.GetToneB);
(e.Dial.e.e   .Dial.DialB);
(e.e.ConReq .e.ConReq.ConReqB);
(e.e.ConnectA.e.ConnectA.ConnectAb);
(e.Answer.e  .e.Answer.AnswerB);
(e.e.ConnectZ.e.ConnectZ.ConnectZb);
(e.e.RingCalled.e.Ring.RingB);
(e.e.Discind.e.Busy.BusyB);
(e.e.CHangup.e.e.CHangup);
(e.e.DetectBusy.e.e.DetectB);
(Hangup.Hangup.e.Hangup.Hangup.Hangup);
(UserTalk.U2serTalk.e.Talk.Talk.Talk).
```

4.4 Validation

Les commandes suivantes permettent de vérifier s'il existe des « deadlock » ou si tous les états sont utlisisés.

```
sync(vecteurs,zp);
dts(zp);
dead:=*-src(*);
proj(6,exist,rsrc(initial)V*);
dts(tr);
notex:=*-exist;
```

Le produit synchronisé a 200 états et 560 transitions. La commande « notex:=*-exist» montre que tous les états sont utilisés. Le modèle permet un retour à l'état initial en tout temps

Les calculs de MEC montrent qu'il n'y a aucun "deadlock" pour ces systèmes de transitions utilisant la liste des vecteurs de synchronisation précédante et tous les états sont utilisés.

4.5 Deux contrôleurs et deux stations

Le deuxième essai contient deux contrôleurs et deux stations. Nous utilisons les mêmes systèmes de transitions que dans le premier essai, sauf que nous ajoutons un contrôleur. Il suffit de copier le contrôleur du premier essai et de les identifier par contrôleur A et contrôleur B. En plus il faut ajouter à Tr les transitions supplémentaires afin de bien identifier le deuxième contrôleur. Les vecteurs de synchronisation changent:

```
synchronization_system vecteurs
<width=7;list=
(UserA      ,UserB      ,ControllerA   ,ControllerB   ,StationA    ,StationB    ,tr)>;
(OffHook    .e          .e             .e             .OffHook     .e           .OffHookA);
(e          .e          .e             .e             .GetTone     .e           .GetToneA);
(Dial       .e          .e             .e             .Dial        .e           .DialA);
(e          .e          .ConReq        .e             .ConReq      .e           .ConReqA);
(e          .e          .RingCalled    .e             .Ring        .e           .RingA);
(Answer     .e          .e             .e             .Answer      .e           .AnswerA);
(e          .e          .ConnectA      .e             .ConnectA    .e           .ConnectAb);
(e          .e          .ConnectZ      .e             .ConnectZ    .e           .ConnectZa);
(e          .e          .Discind       .e             .Busy        .e           .BusyA);
(Hangup     .e          .e             .e             .Hangup      .e           .HangupA);
(e          .e          .CHangup       .e             .e           .e           .CHangupA);
(e          .e          .DetectBusy    .e             .e           .e           .DetectBa);
(e          .OffHook    .e             .e             .e           .OffHook     .OffHookB);
(e          .e          .e             .e             .e           .GetTone     .GetToneB).
(e          .Dial       .e             .e             .e           .Dial        .DialB);
(e          .e          .e             .ConReq        .e           .ConReq      .ConReqB);
(e          .e          .e             .RingCalled    .e           .Ring        .RingB);
(e          .Answer     .e             .e             .e           .Answer      .AnswerB);
(e          .e          .e             .ConnectA      .e           .ConnectA    .ConnectAb);
(e          .e          .e             .ConnectZ      .e           .ConnectZ    .ConnectZa);
(e          .e          .e             .Discind       .e           .Busy        .BusyB);
(e          .Hangup     .e             .e             .e           .Hangup      .HangupB);
(e          .e          .e             .CHangup       .e           .e           .CHangupB);
(e          .e          .e             .DetectBusy    .e           .e           .DetectBb);
(UserTalk   .UserTalk   .e             .e             .Talk        .Talk        .Talk);
```

4.6 Validation

Le produit synchronisé a 140 états et 538 transitions. Il n'y a aucun "deadlock" dans ce modèle et tous les états sont utilisés.Remarquons que cet essai contient moins d'états et moins de transitions que le premier essai, Pourquoi?

Nous observons dans cet essai que les deux contrôleurs sont isolés. Il n'y a aucun échange entre eux en ce qui concerne l'information du « busy » signal ou du fait qu'il y a eu réponse de l'utilisateur. En fait ce modèle montre qu'il y a indépendance entre les deux contrôleurs et qu'il ne

peut y avoir de communication entre des stations étant connectées à différents contrôleurs. Le deuxième essai revient à modéliser deux fois un contrôleur et une station. En fait le modèle POTS ne permet pas l'échange d'informations entre les contrôleurs.

5 Communication entre contrôleurs

Le modèle présenté par [StL93] travaille avec plusieurs stations mais il n'y a qu'un contrôleur qui gère les appels entre les stations. Ce modèle fait la distinction entre quatre types de terminaison d'appel: Avant la signalisation, après la signalisation, après le début de la sonnerie ou bien après l'établissement de la connexion. Le schéma 9 montre la suite de terminaison tel que présenté dans [StL93].

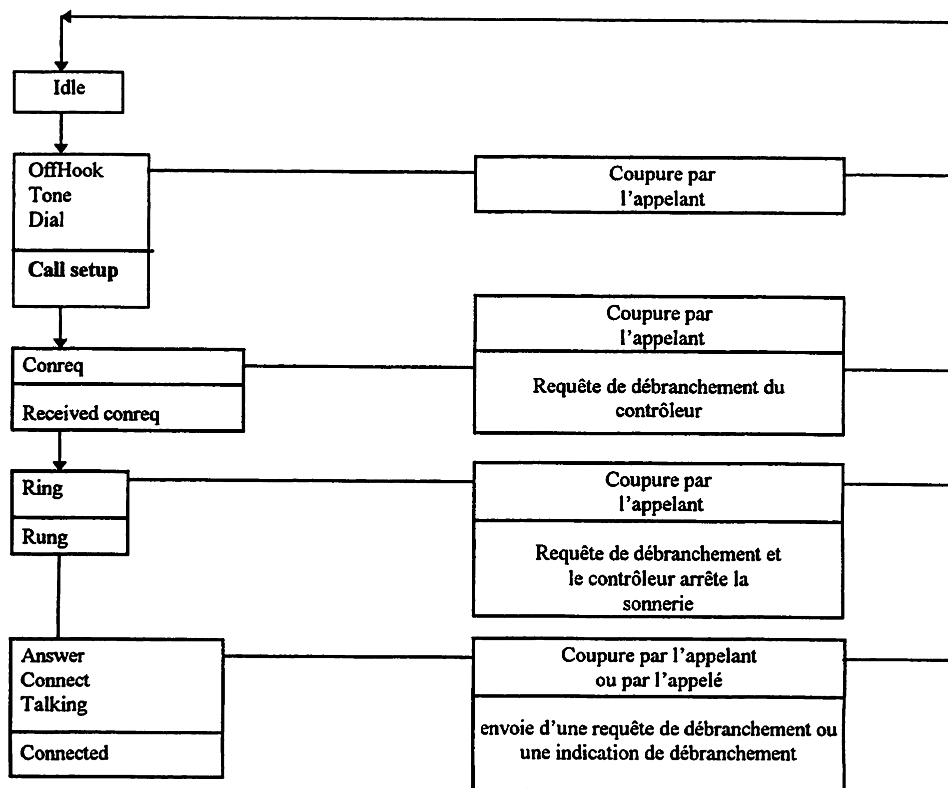


Schéma 9 : terminaisons d'appel

Le modèle présenté en [ArB93] comporte plusieurs similitudes avec celui de [StL93], premièrement ils gèrent l'établissement d'un appel à un même niveau d'abstraction, ils prennent en considération que la station peut gérer le fait d'être libre ou occupé et finalement ils gèrent la terminaison d'appel en considérant l'état dans lequel le système se trouve lors de la terminaison. Ils gèrent aussi les différentes phases d'un débranchement. Le modèle [ArB93] permet de gérer les « time out » pouvant se produire par un processus de surveillance des délais permis pour chaque opération possible du système et prend en compte les échanges de communication entre

deux contrôleurs. Nous retrouvons, en effet, une abstraction élevé de ces échanges par un ensemble de vecteurs qui synchronise les deux contrôleurs.

6 Conclusion

En ce qui concerne MEC, c'est un outil d'analyse qui permet de valider les processus spécifiés sous LOTOS en vérifiant des propriétés nécessaires aux processus. Le transfert de LOTOS vers les systèmes de transitions est simple, rapide et peut être fait de manière automatique. MEC permet d'expérimenter des spécifications en vérifiant rapidement les propriétés nécessaires à ces spécifications. Il permet, dans un cycle de modélisation, d'analyser et de valider la spécification par étapes dans la phase d'édification. C'est un outil pouvant interagir dans un cycle de modélisation.

Bibliographie

[Ar89] A. Arnold, « MEC: a system for constructing and analysing transition systems », Automatic Verification Methods for Finite State Systems, International Workshop Proceedings, 1989.

[Ar90] A. Arnold, « Systèmes de transitions finis et sémantique des processus communicants », TSI., Vol 9, No 3, 1990

[ArB93] A. Arnold, S. Brlek, « Conception d'un système de gestion d'appels téléphoniques: un exemple d'utilisation de méthodes formelles » , Preprint 1993

[BB87] T. Bolognesi et Ed. Brinksma "Introduction to the ISO Specification Language LOTOS". Computer Networks and ISDN Systes 14 (1987) 25-59

[FLS91] M.Faci, L.Logrippo et B. Stepien « Formal Specification of Telephone System in LOTOS: the Constraint-Oriented Style Approcach », de Computer Networks and ISDM systems 21 (1991) page 53-67

[StL93] B. Stépien, L. Logrippo, « Status-Oriented Telephone Service Specification: an exercise in LOTOS style », TR-93-07, Comp. Sc. Dept February 1993

The Validation of Buffer-Based Systems

J.-Ch. Grégoire
INRS-Télécommunications
gregoire@inrs-telecom.uquebec.ca

April 15, 1994

Abstract

Communication protocols usually specify some form of buffer mechanism, together with a management discipline, typically FIFO. The validation of such mechanisms always turns out to be a complex issue for all realistic protocols.

We investigate here the origin of this complexity and propose that it comes from two major sources: the inherent theoretical complexity of buffers and the computational structure used to implement them. We propose techniques that can help to somewhat alleviate the problem and illustrate our point with an example.

1 Introduction

The verification of communication protocols requires the modeling of buffer mechanisms. Buffering is a pervasive technique in protocol design. Incoming or outgoing messages are buffered until they can be processed. We thus find queues between different layers of the protocol stack, typically at a system boundary (i.e. inside and outside the kernel or at the physical interface), or between communicating peers.

Buffers are also used to model some properties of the transmission medium. Indeed, several messages can be in transit at any time, depending on the transmission speed and the store/forward properties of the underlying protocol.

Buffers have a *discipline* of use. It can be for example FIFO, LIFO or priority based. Buffers can be lossy, because of physical problems or also because of overflow. Their size can vary dynamically or they can be static. The combination of possibilities is endless[1].

Sliding window-style buffers are a special class of mechanism that simultaneously handle two related problems. One is to reconstruct message sequences in order on lossy, slow and possibly reordering channels. The other is to improve performance in high latency networks by allowing several messages to be simultaneously in transit.

[1]but bounded

Various researchers have studied the properties of buffers and communication channels, from the theoretical properties to the limits of mechanical verification. In this presentation, we present a summary of some theoretical results relevant to the analysis of buffer based systems. These results show that there are fundamental limitations of buffer based systems and the benefit of having support to simplify their expression. Even then, *validation models* explode in size. We study some techniques that can be used to alleviate the problem. We finally illustrate the use of these techniques in an example.

2 Background

The general context of this work is the identification of computational abstractions suitable for the validation of communication protocols and distributed applications. The validation can be done through mechanical model checking techniques, or with (maybe partially mechanical) theorem proving methods. In this paper, we are primarily interested in the model checking-style verification. Such verification tools usually have two languages: a *modeling language* to capture the system to verify and a *verification language* to express the properties to analyze[2]. We however also present a generic way to look at buffer systems more suitable for theorem proving techniques.

There are wide variations in the size of the state space that can be analyzed with different tools. However, it is hard to capture the model of a buffer-based communicating system in all tools that we know of, with the exception of SPIN and murϕ. Whereas the asynchronous bit protocol is a pervasive example of the use of a tool, the sliding window protocol is much less pervasive. We feel it is symptomatic of the deep complexity of the analysis of such systems.

3 Buffer systems

The structure of communication buffers mechanisms, such as sliding windows, is largely dictated by performance criteria. These can be quite diverse and include:

- minimize congestion,
- optimize throughput,
- minimize latency,
- optimize recovery in the presence of failures,
- balance load,
- etc.

In our context, we are interested in protocol validation and not in performance analysis. Nevertheless, we must consider buffer structures that must satisfy specific performance requirements. Such structures have specific properties which can be verified. The distinction

[2]note that both can be identical, for example in some process algebraic toolkits such as Caesar

between validation and verification is important to note here. We *validate* an implementation of a buffer against an abstract expression of its behaviour. We *verify* directly the expected properties of the implementation, expressed in the same model. For example, we'll validate the FIFO buffer-like behaviour of a sliding window protocol based buffer. On the other hand, we'll verify that any message sent on the communication channel is within the window, using indexing information.

Let us also note that some interesting problems occur in validation because we cannot capture performance characteristics. The modeling of loss is a case in point. Any way we model a lossy channel will result in a potential livelock, where a datum shall never reach its destination. It is a valid cycle in the model, but an (hopefully) extremely unlikely occurrence.

Problems of this variety are typically resolved in verification tools[3] by imposing a *fair* behaviour sequence on the possible evolutions of the model and thus exclude infinite sequences of losses.

The simplest buffer system in telecommunication is the perfect communication channel, that is, a buffer with a FIFO discipline. A more ubiquitous construction is the sliding window buffer which exists under a wide variety of guises depending on the performance criteria that must be met. They will be the basis for our case study.

4 Some theoretical results

A large number of quite interesting theoretical results on the properties of buffers have been collected over the past decade. The properties of infinite and finite communication channels combined with different disciplines have been studied systematically, with the following results.

In [S.84], the authors try to axiomatize different message buffering mechanisms using Temporal Logic. They consider FIFO, LIFO and unsorted buffers. Messages are represented by an alphabet. They show that bounded buffers of any type can be characterized in Propositional Temporal Logic. Unbounded FIFO buffers, on the other hand, cannot be characterized in PTL. The characterization is a formula that is true on the set of sequences specifying the buffer. The characterization, when it exists, can be used to derive an axiom system for the buffer. Interestingly, they prove that LIFO and unsorted unbounded buffers can be axiomatized in PTL.

Wolper, in [Wo86], first proposed that the behaviour of buffers be specified in *data independent way*, that is, the behaviour of the buffer should not depend on the nature of the information stored in it, which can be universally quantified. Data independence extends the class of programs that can be expressed in propositional temporal logic.

Koymans in [Ko87] proposes that the limitation of temporal logic in buffer specification is related to the impossibility of correlating a message that is delivered by the system with a unique message accepted previously by the system. He then observes how different techniques have been used to alleviate the problem. The first one is to use data structures to enrich the

[3] we'll use the expression verification tool independently of the verification vs. validation issue

temporal logic formulation. Another is the introduction of history variables. The other is to have the messages uniquely identified. Wolper's notion of data independence accomplishes just that.

More recently, Sistla and Zuck[SZ93] have studied the properties of a restricted logic called RTL, which is strictly less expressive than full temporal logic (e.g. it is missing a next operator). They have shown that axiomatizing the behaviour of FIFO or unordered message buffers in RTL is decidable in exponential time.

Abdulla and Jonsson[AJ93] have studied finite state machines communicating over unbounded FIFO, lossy channels. They have shown that reachability, safety over traces and eventuality properties can be decided, even though they are not decidable for unbounded, non lossy FIFO channels.

Manna and Pnueli[MP92] use a buffer to illustrate the expression of several types of temporal properties. They use a linear temporal logic with past time operators and illustrate the expression of safety and liveness-class properties. On the safety side, the *absence of unsolicited response* can be viewed as a causality constraint: we express that for something to be read from a buffer, something must have been previously put into the buffer. We can also express that the output should not be duplicated, that the ordering matches the expressed discipline of the buffer (FIFO, in this case). A lifeness-style property is the *ensured response*: the guarantee that something put in the buffer will eventually be delivered.

They furthermore present different forms of specification of buffers, using quantified temporal logic and also of channel based message passing system. Once again, their specification is based on auxiliary or history variables.

5 Computational buffers

These results show that we cannot specify a universal FIFO-style communication channel in a pure logical way, or else only in a restricted way. When the capacity of the channel may vary over a finite domain, the complexity of the expression explodes dramatically with the size of the domain. These results have an important impact on verification tools. They show the difficulty of modeling buffer systems. The axioms required become quite large and complex for systems of relatively small size. The state space actually suffers a similar explosion.

To express system models we can use a technique similar to that used in formalization and use an auxiliary structure to capture implicitly some of the buffer's state behaviour. This is done for example in the SPIN tool. The tool has a generic built-in channel data structure which can be instantiated to the proper size. The specification is not generated explicitly but computed on the fly through the exploration of the state space.

This technique is quite useful to model the behaviour of communication channels. Combined with nondeterministic selection, it is also possible to express arbitrary loss in the channel. It is important to notice that we still have an explosion in the size of the state space, which is unavoidable, but not in the expression of the model of the channel-based communication system. However, the validation of communication systems requires the

modeling of customized buffer mechanisms, such as sliding windows, and we lose the benefit of the built-in structure.

Modeling sliding windows is done most easily in an imperative implementation style, that is, using the same computational structure one would write in - say - C. Once again, it would be a computational model from which the state space could be generated on the fly, rather than an axiomatization which explicitly describes the whole state changes. This representation however introduces new questions with regard to validation. A computational model is further away from an abstract specification and besides the validation of its properties, we must also *verify* the soundness of its own structure, i.e. not only what is done, but the way it is done.

A simple sliding window mechanism is thus described in [Ho91], using arrays, indexes and modular arithmetic. The properties validated reflect closely the mechanism, such as guaranteeing that message retransmitted are within the window: this is a verification of the soundness of performance structures rather than validation of FIFO behaviour.

6 General optimizations

Let us now review some abstractions that can be used to bring down the size of a buffer-based model and make verification more manageable. We can readily find an explanation for the state space explosion in a basic property of validation models: all the possible execution interleavings are generated, for all possible values transmitted.

Orthogonality The validation of applications communicating over buffers is usually done in orthogonal fashion. The buffer mechanism is first validated with minimal assumptions over the customer processes' behaviour. Then the two processes are considered, with a perfect communication channel.

Size reduction The most obvious simplification in the model is to reduce the size of the window to the smallest value that still preserves the same behaviour patterns. The buffer algorithm is usually size independent. The difference the size makes is typically in the amount of interleaving of the communicating application processes and does not have an impact on the mechanics of the buffer itself. The reduced model still allows one to verify the soundness of the mechanics of the buffer, such as boundary checking and proper embedding of the information sent within the window.

Three is typically an interesting size, since it includes boundaries and an internal value.

Data independence The behaviour of the buffer is data independent, as well as size independent. We should not have to include data representation in the buffer. However, whereas we can easily study the mechanism itself in a data independent way, it is not the case for the application, because the transmitted data may influence the behaviour of (higher level) customer processes. Therefore, whereas we can abstract out data information in the

analysis of the mechanics of the buffer, we still have to worry about its presence in an ideal communication buffer.

Symmetries The behaviour of a buffer can be described not with regard to buffer locations but with regard to data categories within the buffer, e.g. sent once, resent, acknowledged or not acknowledged. Within the categories, we do not need to distinguish between the different elements, since the behaviour will be similar, with the exception of the possible buffer boundaries adjustments.

Synchronism If applications are interleaving degree independent, we can consider them simply in a synchronous framework and use *rendez-vous* for exchange rather then buffers.

Computational abstractions It is sometimes possible to remove some computational information from the state model while still generating all the behaviour required to validate the properties of the model[Gr94]. This is done by hiding the details of the computation from the model, keeping only results as they influence state variables. This can be realized rather easily in a tool where the behaviour model is computed on the fly from an algorithmic description, as is the case with the PROMELA/SPIN tool, rather then being described exhaustively. The overall effect is to reduce the number of states to explore to prove a property.

Generalization We can also sometime validate only a restricted behaviour, and then generalize it to more complex structures. A common example is the validation of a one-way communication channel, and its generalization to a bidirectionnal channel.

As we abstract out information, we move from a verification problem to a validation problem. It is important to understand the semantic impact of the abstraction to see that is is meaningful (does what we want), complete (always applies over the domain) and sound (preserves the semantics wrt validation of other properties). It is not always possible to formally guarantee that these criteria are met.

Abstractions may however be the only way to cut down a model to a size amenable to validation. This is true not only of buffer based systems, but of verification models in general.

7 A generic structure for buffer based transmission

7.1 Introduction

We now introduce a generic structure for buffer-based transmission. The term generic indicates that we want to capture all the general properties of such transmission without involving performance concerns. Optimized structures should be defined only as refinements

of the generic structure. In what follows, we consider several processes. Processes executes functions or operations and receive events. An operation creates an event for another process. An event may carry data.

7.2 The model

Let us consider a protocol layer which attempts to realize a reliable link between two points over a lossy channel. We first assume that data is kept in a buffer of a fixed and identical size at both sides. Data is given to the layer in some order on one side and recovered from it in an appropriate order, which may or may not be the same on the other side. The (conceptual) link may however also be completely lost, because of physical failures or untolerable transmission noise.

We have a `sender-user` process, a `sender` process, a `channel` process, a `receiver` process and a `receiver-user` process.

`sender-user` can execute a `send` operation and receive a `room available` (`rm`) event, which signals that there is room available in the buffer, and how much. `receiver-user` receives a `new data` (`nd`) event together with the data. Indeed, the receiver side does not know *a priori* when to expect a new message.

For the purpose of transmission, we need only to consider the buffer as a *bag*, i.e. without an ordering structure. We only require that every datum in the transmission system be uniquely identified. Any structure that is required for transmission will be explicitly defined in a refinement step.

`sender` executes a `transmit` operation to send a bag over the channel. It also receives two events: a `retry` event which triggers the transmission of the bag, and an `ack` event, which identifies the fraction of the bag sent that has been received. In the case of a lossy channel, `ack` behaves like a random function, returning any fraction of the bag. With a perfect channel, `ack` would return the whole bag. If the physical link is broken, `ack` is never received. `data-to-send` (`dts`) is the event resulting from the execution of `send` by `sender-user`.

`channel` receives `data-to-transmit` (`dtt`) event following the excutin of `transmit` and does either nothing, or invalidates parts of the bag and executes a `deliver` operation for `receiver`. Its behaviour is similar in the other direction of communication.

What is transmitted following an `ack` is some subset (b') of the bag sent previously (b). The subset is identified by a `selection` function and `ack`. The selection function must however be such that $b' \subseteq b$. If we assume that the probability of loss is data and position independent, and not one, we'll eventually have a copy of the original buffer on the receiver side. At this stage, we can signal that the send buffer is empty, that the receive buffer is full, fill up the first, empty the other and renew the process.

`channel` behaves similarly in either direction of transfer. It either drops the bag, transmits it unchanged, or randomly modifies its content. This behaviour applies to both data or acknowledgments alike.

`receiver` receives a `data` event with the arrival of a potentially partially corrupted bag

b_r. An `update` function will create the new value of the bag received so far. A `notify` operation transmits the acknowledgment of b_r back to the sender side. If the local receiver bag is complete, `receiver` executes an `accept` operation which transmits the received data to `receiver-user`.

Obviously, in this model, the progress depends on the two events: `retry` or `ack`. There is a possibility of deadlock if the `ack` is lost - a typical problem. This is when `retry` has to be used. The typical way of generating this signal is with a timeout. In this case, the previous partial buffer will be resent, without change. Let us note that the use of timeouts typically leads to the duplication of data, although such behaviour was already allowed in this general model.

Let us notice that we haven't made any assumption of size anywhere so far in our informal reasoning. We haven't made any restriction on the identification and the selection of the subset. What we have is a general framework that can be refined for different protocols. But any mechanism which satisfies these properties is guaranteed to transfer information. Also, we make the assumption that the information is dropped, but not changed by the transfer. Once again, this is easy to guarantee in practice (or at least the probability of non-detection of the corruption can be kept low enough) with error detection codes. Then, the detection of corruption amounts to a loss.

In parallel with the transmission, but independently, although this introduces implications on the multiplexing of the channel which can be resolved at a lower layer, we will also have a `keep alive` (`ka`) message exchange meant only to test the channel. This can also be done from either end. Loss of the physical link results in a the transmission of a `link lost` (`ll`) event to the higher layer on either side. Practically, this mechanism can be used to generate the `retry` event for `receiver`.

This generic model captures all there is to know about transmission over a lossy channel. We need only to refine the following elements:

- some way to identify any datum, to add them and remove them from bags;
- a feedback event (`ack`) to indicate what has been received so far;
- another event (`retry`) to compensate for the loss of the feedback event;
- a function (`selection`) to extract a subset that hasn't been acknowledged;
- a function (`update`) to identify newly received datums.

And, of course, the operations we have listed.

The events are required to make sure there is progress. The functions must guarantee that the amount of information sent is monotonically decreasing, i.e. with $|b'| = |b|$ after a `retry` and $|b'| < |b|$ after a `ack` (if all acknowledgment information is lost, the whole event is considered lost). For `receiver` we similarly have that $|B_r|$ is monotonically increasing.

We see there is actually little to prove about the sound behaviour of a buffer based system: the existence of a chain of events, and the monotonicity of the functions. The physical channel must guarantee that data eventually is transmitted, but this is a requirement of the physical system rather than something that can be verified.

7.3 Identification

As mentioned, we must guarantee that every individual datum be uniquely identified for the bag structure to be effective to update the subset to send and the subset received so far. The unique identifier also identifies and removes duplicates.

It may seem a priori difficult to find such an identification that could be confined to a layer only. However, there exists a well known and well studied solution to this problem, which is the numbering scheme for the sliding window protocol. We thus know that we only need a set of numbers at least twice as large as the capacity of the sender's buffer, i.e. $2 \times max(|b|)$. Identifiers can be recycled implicitly with modular arithmetic. This, of course, requires the buffers to be bounded, or, at least, to have a statically defined upper bound.

Note that the unique identifier property and the mechanism used by the sliding window protocol help to trivially realize a FIFO behaviour for the transmission. We just need to take datums out of the system in increasing identification number.

7.4 Incremental transmission

The model we have described made the assumption of a full update of `sender`'s buffer. The buffer is refilled only when it is empty. This assumption makes it easier to see that the transmission eventually terminates with a copy of the original buffer on the `receiver` side.

In practice, however, we want to be able to send data incrementally, and fill the buffer as soon as there is room available, typically to optimize the use of the bandwidth. This means that the buffer may be modified between two receptions of `ack` or `retry`. This does not however invalidate the requirement that the `selection` function be monotonic. We need only to guarantee that

1. a new datum does not replace one that hasn't been acknowledged or sent so far;
2. a new datum will not prevent older datum from being transmitted;
3. a new datum is identifiable.

We must also assume that room is made on the receiver side in similar fashion. We know however that room cannot be made available on `sender` before an `ack` is received from `receiver`. If buffers are the same size, this condition is satisfied trivially.

The first problem is pure bookkeeping. The second is implicitly resolved with the generic transmission procedure which does not discriminate between datums. The third problem is part of the more general problem already mentioned.

We see that incremental transmission is easily achieved, and what needs to be verified is the buffer update process, to guarantee that older datums are not overridden.

7.5 Other problems

We did not consider here the problem of starting and synchronizing the different processes. Protocol elements could easily be added to this effect, but communication structure used

in practice tends to be implementation independent and would not contribute any useful analytic property. We did not similarly consider cancellation and reset signals.

7.6 Refinements

There is a large variety of refinements that can be considered. It is possible to modify the size of the buffer if the error rate is too important, we can guarantee "at-most-one" delivery of messages...

The coding of acknowledgments is also left open: it can be incremental or complete. It is also possible that the bag sent contains only a single datum - which is the more likely case.

Resending can be performed in any order depending on the characteristics of the channel.

Some refinement could require new signals, for example to temporarily suspend the transmission pending the availability of buffer space. We would then have to study there interaction with the other signals, and make sure that, with some assumption of fairness, the transmission function will be repetitively executed.

7.7 Conclusion

We have described a generic behaviour model for buffer-based transmission systems. This model achieves the requirement that any data transmitted to `sender` will eventually be recovered from `receiver`.

In the next section, we'll show how this model can be extended for some performance concern, while preserving its original properties.

8 Example: Validation of a buffer based protocol

8.1 Introduction

The Radio Link Protocol (RLP) is a new link layer protocol under study by a standardisation committee[RLP]. It is dedicated to radio application with a focus on the optimization of the number of retransmissions of lost data, to guarantee at-most-once delivery.

The protocol is built around a classical sliding window mechanism, with a couple of additions. First, feedback from the receiver is given in the form of a bitmap which indicates exactly what has been received. Second, an extra *send sequence number* (ssn) is associated to every packet in the send window. This number is increased whenever a packet is transmitted or retransmitted. For every acknowledged packet, only unacknowledged packets with a lower ssn may be resent without fear of duplicated delivery.

These mechanisms are meant to guarantee that no packet will be received twice, and to optimize the use of the limited bandwidth.

The verification of the protocol requires the construction of a model of the extended sliding window mechanism. A first validation model using PROMELA/SPIN has been used to check the basic flow of the protocol, namely a deadlock when the window is full and all

acknowledgments are lost. A version of the protocol that resolves the problem exists and has been validated.

The verification model is quite large, for several reasons. The model computes exactly the sliding window mechanism, which has a combinatorial structure. Added complexity comes from the use of bitmaps, but mainly from the ssn. This model is however closely related to a possible implementation.

8.2 An abstraction-based approach

We have investigated how an alternative model based on the combination of functional abstraction, data abstraction and symmetry could be developed for the RLP.

Let us have the following considerations.

- The transmission window is of maximal size S. Practically, in this case, the size of the window is constant, but this allows us to handle startups and end of transmissions in the same framework.
- The behaviour of the protocol does not depend on the index of the element to retransmit. It is just used to select which item to send, not how it is going to be sent.
- Any transmitted or retransmitted packet may be lost, independently of its sequence number or sender send number.
- The number of retransmission of information is irrelevant. It is however bounded by the size of the window, at least for the version without preemption.
- Startup and end are symmetrical.

With these considerations in mind, we have built a new model where the window is functionally abstracted. On the basis of the information received as part of the acknowledgment, we consider a window to be the union of three non-overlapping sets: the set of packets with a lower ssn that have been received, the set of packets with a lower ssn that have not been received and the set of packets with a higher ssn. These sets are not necessarily compact, but, since we do not rely on the values of the indexes, but only on the cardinality of the subsets, we consider that they hold sequentially numbered packets. The sets have their own context and they hide the management of the ssn information. Transfer of data is done through server queues. Data is removed from a queue and non deterministically put into another one back to the sender, or lost.

Whenever there is a new item to send, or when an acknowledgement is received, the subset of items to transmit is rebuilt, and we proceed with transmission. We model the number of packets and of acknowledgments received. The modeling is over when all the ssn have been used. There must therefore be enough of them to guarantee that an entire window may be retransmitted once, to generate all the possible combinations of set cardinalities in the presence of losses. Greater values do not bring any more information. The split of the set and the guarantee that a packet is not repeated is given by the set decomposition, and becomes an invariant of the functional abstraction. What is left to verify, for the basic

protocol, is the possibility of deadlock. This is done trivially, since all the details of the implementation of the window have been removed, and the state vector shrunk accordingly.

The practical result of this exercise was the switch from a stochastic analysis of the protocol using the supertrace mode of SPIN to a deterministic analysis in a small state space.

8.3 A generic model approach

The structure of the protocol fits within our general model. The critical refinement we must worry about is the guarantee of single delivery. Let us discuss how this can be established as a refinement of the `selection` function.

Let us remember one key property of the physical communication link: it does not reorder information. Therefore, considering every datum is sent separately, and acknowledged as soon as it is received, Once we receive an acknowledgment for a message, either all preceding ones have been received, or some have been lost, or the acknowledgment of reception has been lost. It is for this reason that any acknowledgment reacknowledges previously received messages. Therefore, once we receive an acknowledgment, we know exactly what can be resent, if anything, provided we kept track of the send order.

To establish this property, `selection` uses a queue, which is rebuilt whenever an acknowledgment is received according the the following rules, reading sequentially from the old queue:

1. as long as there remains a datum in the queue that is in the acknowledged list,
   (a) discard acknowledged datums,
   (b) resend unacknowledged datums and append them to the new queue;
2. as soon as all acknowledged datums have been removed from the queue, append the rest of the queue to the new queue.

Out of `selection`, only new datums can be sent and added to the queue. We can derive the following invariants:

- every datum in the queue is unique
- no element is resent unless it is not acknowledged and another datum sent afterwards is acknowledged.

Reasoning recursively on the head of the queue: if is it acknowledged, we do not need to resend it. Otherwise, either there is another message sent later that has been acknowledged, and we need to resend it, or not and then this is also true of the rest of the queue. No element is put in the new queue that isn't already in the old queue, or that is newly sent. The second invariant, combined with the no-reordering property of the channel guarantees that a message can only be delivered once.

9 Conclusions

We have summarized general characteristics of the validation of buffer based communication systems, and offered new insight on the complexity of their expression and their analysis.

Buffers are an important part of most industrial protocols, and contribute significantly to the explosion of the state space. We feel that too little has been done to understand this problem and to propose techniques to alleviate it in a way that does not jeopardize verification.

We have shown two ways to approach verification of large protocols. The first one is to use various forms of abstractions to reduce the state space. The other is to use a generic structure that satisfies the general transmission property and refine it to verify the specific properties we are concerned with. This latter may require using specific theorem proving technique rather than generic model checking, but focuses the problem on specific items, instead of the whole protocol.

We have experimented with the first way using a customized version of SPINẆe are investigation the use of Lamport's Temporal Logic of Actions[La91] for the second.

Acknowledgments.

Thanks are due to Michael Ferguson and Bob Johnston for some thought provoking remarks on the nature of verification. M. Alipour, E. Gauthier, A. Lemoine and B. Sanscartier have contributed to experiments on the analysis of the sliding window buffer with different tools. The queue mechanism for RLP was originally proposed by E. Gauthier, for a different design.

References

[AJ93] P. Abdulla and B. Jonsson, *Verifying Programs with Unreliable Channels*, Logic in Computer Science 1993, Computer Science Press, IEEE, pp. 160–170, 1993.

[GS94] J.R. Gallardo and J. Sanchez, *Omission and Ambiguities in CCITT Recommendations Q.921*, IEEE Communications Magazine, Vol. 32, no. 2, pp. 88–94, 1994.

[Gr94] J.-Ch. Grégoire, *Computational Abstractions in PROMELA/spin*, submitted for publication, 1994.

[Ho91] G. Holzmann, *Design and validation of computer protocols*, Prentice Hall Software Series, 1991.

[Ko87] R. Koymans, *Specifying Message Passing Systems Requires Extending Temporal Logic*, in "Temporal Logic in Specification", Lecture Notes in Computer Science 398, Springer Verlag, pp. 213–223, 1987.

[La91] L. Lamport. *A Temporal Logic of Actions*, DEC SRC Research Report No. 57, 1990.

[MP92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992.

[Q921] ITU, *Recommendation Q.921 - ISDN Data Link Layer Specification*, 1992.

[RLP] Al Sacuta (Chair), *RLP1 SDL Diagrams and Variable List (Revision 6)*, TR45.3.2/94.02.08.03, TDMA Cellular Systems Data Services Working Group, Feb. 8, 1994.

[S.84] A.P. Sistla, E.M. Clarke, N. Francez and A.R. Meyer, *Can Messages Buffers Be Axiomatized in Linear Temporal Logic?*, Information and Control, Vol. 63, pp. 88–112, 1984.

[SZ93] A.P. Sistla and L.D. Zuck, *Reasoning in a Restricted Temporal Logic*, Information and Computation, vol. 102, no. 2, pp. 167–195, 1993.

[Wo86] P. Wolper, *Expressing Interesting Properties of Programs in Extended Temporal Logic*, Principles of Programming Languages, ACM Press, pp. 184–193, 1986.

Vérification des chronogrammes hiérarchiques à l'aide de "CCS + contraintes"

Bachir BERKANE, Eduard CERNY

Dép. d'IRO, Université de Montréal, C.P. 6128, Succ. A
Montréal, Québec, H3C 3J7 Canada
{berkane, cerny}@IRO.UMontreal.CA

Résumé. Cet article développe une approche de vérification formelle des chronogrammes hiérarchiques (CHs). Nous décrivons le comportement qualitatif d'un CH (sans temps métrique) à l'aide d'une expression de l'algèbre des processus communicants de Milner (CCS). Ensuite, nous transférons la dimension temporelle quantitative des contraintes temporelles d'un chronogramme vers le système de transition dénoté par l'expression CCS. Le modèle obtenu est appelé *graphe de régions relatives*. Enfin, nous montrons que la vérification de la consistance d'un chronogramme hiérarchique consiste à parcourir l'espace d'états de son graphe de régions.

Mots-clés : Chronogrammes hiérarchiques, "CCS + Contraintes", Vérification

1 Introduction

La vérification des propriétés temporelles quantitatives d'un système matériel numérique est un moment crucial lors du processus de conception. En effet, un système qui ne satisfait pas des contraintes de délais peut remettre en cause les phases de son développement. Durant les dernières années plusieurs travaux ont traité le problème de la vérification des systèmes temps réel où des délais entre les entrées et les sorties ont été considérés dans la spécification de ces systèmes. Dill [Dil 89] et Lewis [Lew 89] ont proposé une méthode de modélisation et de vérification des circuits asynchrones par l'addition d'un système d'horloges à l'automate d'états finis décrivant le circuit. Une logique temps réel arborescente interprétée sur ces automates a été définie et des algorithmes de model checking ont été présentés dans [ACD 90, HNSY 92]. Dans [BD 91] des réseaux de Petri temporisés ont été utilisés pour la modélisation et la vérification des systèmes temps réel. Une approche similaire a été développée par Yoneda *et al.* [YSSC 93] pour examiner les ordres partiels dans le dessein de limiter le problème d'explosion d'états. D'autres travaux ont abordé des aspects particuliers de la vérification (voir par exemple : [MD 92], [AHBB 93], [AB 92]).

Khordoc *et al.* [KDC+ 93, KC 93] ont proposé une méthode de spécification des interfaces matérielles basée sur des chronogrammes hiérarchiques. La méthode consiste à spécifier les comportements "entrées/sorties" récursivement à l'aide des chronogrammes. Un chronogramme-feuille est composé d'un ensemble de ports où les actions ont lieu. Les temps d'occurrence des actions sont liés par un ensemble de relations temporelles. A partir d'un nombre d'opérateurs de composition, un chronogramme hiérarchique est formé. Cet article comble la lacune entre cette

méthode de modélisation et un cadre de spécification et de vérification formel. Nous montrons tout d'abord que la sémantique opérationnelle de l'algèbre de Milner (CCS) [Mil 89] convient à interpréter les comportements non temporisés des chronogrammes hiérarchiques. Plusieurs travaux ont étendu les algèbres de processus (à la CCS) au temps dense par l'addition d'un ensemble d'opérateurs temporels. Une étude comparative peut être trouvée dans [NS 91]. Ces extensions sont mal adaptées à la modélisation des contraintes temporelles des chronogrammes.

Pour tenir compte du temps métrique des contraintes temporelles nous nous sommes inspirés des travaux développés dans [BD 91, YCSS 93] pour définir la sémantique opérationnelle "temporelle" (dans le sens dense) associant un modèle, appelé graphe de régions relatives, à un chronogramme. Chaque noeud du graphe est un couple formé (1) d'une expression CCS et (2) d'un ensemble de contraintes temporelles décrivant des relations temporelles entre un ensemble d'actions.

La vérification de la consistance d'un chronogramme (une spécification d'interface) consiste à s'assurer que tout événement peut avoir lieu en cohérence avec les contraintes temporelles. Cette vérification se ramène à parcourir l'espace d'états du graphe de régions associé au chronogramme; l'exploration du graphe est réalisée sans le construire (exploration "à la volée").

Le reste de l'article est organisé comme suit : La section 2 décrit la méthode de spécification des chronogrammes à l'aide de "CCS + contraintes", dans laquelle un graphe de régions relatives est associé à un chronogramme. La section 3 présente la méthode de vérification. L'application de la méthode sur un exemple se trouve dans [BC 94]. La section 4 conclut l'article par la présentation de quelques perspectives de travail.

2 Spécification des chronogrammes à l'aide de "CCS + Contraintes"

Nous définissons dans cette section une méthode de spécification des chronogrammes hiérarchiques à l'aide de "CCS + Contraintes" désignée par T^cCCS (de l'anglais, Time-constained CCS). Nous donnons d'abord un résumé succinct de la méthode de spécification de Khordoc *et al.* et nous montrons comment le comportement non temporisé (qualitatif) d'un chronogramme peut être spécifié à l'aide de CCS. Nous montrons par la suite comment le temps métrique peut être ajouté à CCS pour saisir l'information métrique des contraintes temporelles. Nous décrivons la sémantique opérationnelle temporelle de T^cCCS permettant d'associer à une expression T^cCCS un modèle appelé "*graphe de régions relatives*". Enfin, pour identifier les expressions T^cCCS nous définissons une relation d'équivalence.

2.1 Chronogrammes hiérarchiques : CHs

Un *chronogramme-feuille* (CF) est composé d'une part, d'un ensemble fini de *ports* d'entrée et de sortie où des actions atomiques (événements) sont exécutées et d'autre part, d'un ensemble fini de relations temporelles entre les actions appelées *contraintes*. Chaque CF possède deux actions particulières : *début* et *fin*. L'action "début" est exécutée avant toute autre action et l'action "fin" est exécutée lorsque toutes les actions du CF ont été exécutées.

Une *contrainte temporelle de base* ou TCons est de la forme "$m \leq t_{l'} - t_l \leq M$", où m et M sont des entiers ($m \leq M$) et $t_{l'}$, t_l sont les variables temporelles à valeurs dans R+ (les réels positifs) définissant respectivement le temps d'occurrence des deux actions l' et l. Une contrainte temporelle définit un intervalle $[m, M]$ et spécifie que la distance temporelle entre les deux actions

l et l' est à l'intérieur de l'intervalle $[m, M]$. Si $m \geq 0$, la contrainte définit une relation orientée appelée "*précédence*".

Une contrainte temporelle est caractérisée par son *type* : *assertion*, *réaction* ou *propriété*. Les contraintes d'assertion définissent des relations entre les actions d'entrée ou des précédences entre les actions de sortie et les actions d'entrée. Les contraintes de réaction définissent des relations orientées des actions d'entrée ou de sortie (désignées par *actions source*) vers les actions de sortie (désignées par *actions puits*). Enfin, les contraintes de propriété (notées $prop_{\{l,l'\}}$) sont des propriétés temporelles de sûreté que le système doit satisfaire durant tout comportement. Les sous-ensembles d'actions d'un CF et d'un CH liées par des contraintes de propriété sont notés respectivement $Prop_{leaf}$ et $Prop$, $Prop = \bigcup_i Prop_{leaf_i}$.

Avec le dessein d'exprimer des relations temporelles complexes, trois opérateurs de composition ont été définis: *"Earliest"*, *"Latest"* et *Conjonctif*.

Les opérateurs "Earliest" et "Latest" combinent n contraintes de base du type réaction définissant le même intervalle $[m, M]$ et ayant la même action puits. L'opérateur "Earliest" (resp. "Latest") spécifie que l'action puits a lieu au plus tôt m unités de temps et au plus tard M unités de temps après l'occurrence de la première (resp. la dernière) action source. Le temps d'occurrence t_l de l'action puits l est défini comme suit :

$$Earliest_{[m,M]}(l_1, \ldots, l_n) \Leftrightarrow \underset{i=1\ldots n}{Min}(t_{l_i}) + m \leq t_l \leq \underset{i=1\ldots n}{Min}(t_{l_i}) + M$$

$$Latest_{[m,M]}(l_1, \ldots, l_n) \Leftrightarrow \underset{i=1\ldots n}{Max}(t_{l_i}) + m \leq t_l \leq \underset{i=1\ldots n}{Max}(t_{l_i}) + M$$

où t_{l_i}, $1 \leq i \leq n$, est le temps d'occurrence de l'action source l_i. (Signalons que ceci est plus restrictif que la définition donnée dans [KC 93, MD 92], mais c'est suffisant en pratique.)

L'opérateur *Conjonctif* définit une collection de contraintes de base du type assertion. Pour toute paire d'actions (l, l') liée par une contrainte dans la collection, il existe une autre contrainte dans la collection liant au moins une des deux actions l ou l'. Les contraintes de la collection sont conjonctives, c.-à-d. que toutes les contraintes doivent être satisfaites simultanément.

Les chronogrammes peuvent être composés à l'aide des opérateurs de *concaténation*, de *composition parallèle*, de *choix non déterministe* et de *boucle*. La description de ces opérateurs à l'aide de CCS est donnée dans la sous-section 2.2.3.

2.2 Spécification du comportement qualitatif des chronogrammes à l'aide de CCS

Nous montrons ici comment CCS peut être utilisée pour décrire les comportements qualitatifs (non temporisés) des chronogrammes, c'est-à-dire les ordres partiels entre les actions engendrés par les différentes contraintes temporelles entre les actions.

2.2.1. Quelques mots sur CCS

CCS (de l'anglais, Calculus of Communicating System) [Mil 89] est un environnement mathématique pour la description et la vérification des processus communicants. Un processus (appelé *agent*) est spécifié par le biais d'une expression algébrique composée de quelques opérateurs et d'un ensemble fini d'actions atomiques. Un agent exécute un ensemble d'actions d'entrée et de sortie. L'ensemble des actions exécutées est noté $Act = N \cup \overline{N} \cup \tau$, où $N = \{a, b, \ldots\}$ et $\overline{N} = \{\bar{a} \mid a \in N\}$ sont respectivement des ensembles d'étiquettes (d'actions d'entrée) et de coétiquettes (d'actions de sortie) et τ est une action interne (action de communication). Les

éléments de $N \cup \overline{N}$ sont notés l, l',... et les éléments de Act sont notés α, β, ... Les agents de CCS notés A, B, ... sont définis par la syntaxe suivante :

$$A := Nil \mid l\,.\,A \mid A + B \mid A \,|\, B \mid A \backslash L' \mid A_{[f]}.$$

- *Nil*: est un processus terminé (inactif).
- *Préfixage* : "$l\,.\,A$" exécute l'action l pour se comporter comme A.
- *Choix non déterministe* : "$A + B$" se comporte comme A ou B ; le choix est non déterministe.
- *Composition parallèle* : "$A|B$" peut exécuter les actions de A et de B ensemble. Les agents A et B peuvent communiquer entre eux ; dans ce cas "$A|B$" exécute l'action interne τ.
- *Restriction* : "$A\backslash L'$" se comporte comme A, mais ne peut pas communiquer avec d'autres agents par le biais des actions l' et $\bar{l}'$, $l' \in L'$.
- *Réétiquetage* : "$A_{[f]}$" est l'agent A ayant ses actions renommées par la fonction f.

Un agent CCS A dénote une machine abstraite caractérisée par le système de transition étiqueté $(S, Act, \rightarrow, A)$ où S est un ensemble d'états correspondant à un ensemble d'expressions CCS, $\rightarrow\ \subseteq S \times Act \times S$ est la relation de transition, et l'expression A définit l'état initial. On trouvera dans [Mil 89] la sémantique opérationnelle de CCS associant à une expression un système de transition étiqueté.

2.2.2. Chronogrammes-feuilles

Nous introduisons ici, en plus des actions du chronogramme, les actions auxiliaires p_i, go_i, end_{port}, ... et $\overline{p_i}$, $\overline{go_i}$, $\overline{end_{port}}$, ... ($i \in I$, où I est un ensemble d'indices). Le comportement de chaque port d'un CH est décrit par le biais d'un agent CCS exécutant les actions du port et quelques actions auxiliaires. Nous définissons un agent de *synchronisation* noté *Sync* pour chaque opérateur de contrainte liant un ensemble d'actions source avec une action puits. La notation "$l^k.\ A$" représente le préfixage de l'agent A par k ($k \geq 0$) occurrences de l'action l ; $l^0.\ A \equiv A$.

Définition 2.1. Nous définissons deux types d'agents de *synchronisation Esync* (pour l'opérateur "Earliest") et *Lsync* (pour l'opérateur "Latest"):

$$Esync_m \stackrel{def}{=} p_m\,.\,\bar{s}\,.\,\overline{go}_m\,.\,Nil \mid s\,.\ p_m^{k-1}\,.\,Nil \qquad (k \geq 2)$$

$$Lsync_m \stackrel{def}{=} p_m^k\,.\,\overline{go}_m\,.\,Nil \qquad (k \geq 1)$$

L'agent *Esync* est composé de deux sous-termes pour permettre aux agents décrivant les ports, où les actions source seront exécutées, de continuer d'évoluer après avoir exécuté une action source. Lorsqu'un ordre entre les actions est défini par l'opérateur *Conjonctif* nous utilisons un agent *Lsync* pour le spécifier.

Définition 2.2. Etant données m actions l_1, ..., l_m. L'agent *Port_beh* décrivant un port est défini comme suit :

$$Port_beh \stackrel{def}{=} go_1^j\,.\,l_1\,.\,\bar{p}^{k1}\,.\,\ldots\,.\,go_m^j\,.\,l_m\,.\,\bar{p}^{km}\,.\,\overline{end}_{port}\,.\,Nil,$$

où $j \in \{0, 1\}$. L'exécution de l'action $\overline{end}_{port}$ indique que toutes les actions du port ont été exécutées.

Définition 2.3. L'agent *Kernel* décrivant un CF sans les actions "début" et "fin" est la composition parallèle des agents décrivant les ports du CF ($Port_beh_j$, $1 \leq j \leq n$) et les agents de synchronisation ($Sync_i$, $1 \leq i \leq x$):

$$Kernel \stackrel{def}{=} (Port_beh_1 \mid ... \mid Port_beh_n \mid Sync_1 \mid ... \mid Sync_x) \setminus \{\{p_i\}, \{go_i\}, \{s_i\}\}$$

Un agent de synchronisation associé à un opérateur de contrainte communique à travers la paire d'actions (p_i, $\overline{p_i}$) avec les agents décrivant les ports où les actions source sont exécutées et avec l'agent décrivant le port où l'action puits est exécutée à travers la paire d'actions (go_i, $\overline{go_i}$). A chaque communication, l'agent *Kernel* exécute l'action interne τ.

Enfin, l'agent *Leaf* décrivant un CF est défini comme suit :

$Leaf \stackrel{def}{=} begin . (Kernel \mid End) \setminus \{end_{port}\}$, où $End \stackrel{def}{=} end^n_{port} . \overline{end_{leaf}} . Nil$

L'agent *End* communique avec l'agent *Kernel* via les actions end_{port}. Il exécute l'action end_{leaf} lorsque toutes les action de l'agent *Kernel* ont été exécutées.

2.2.3. Composition

Nous donnons à présent la définition des opérateurs de composition de CHs à l'aide de CCS. Dans ce qui suit nous considérons que chaque agent CCS décrivant un chronogramme CH_i a l'action notée $begin_i$ comme action "début" et l'action notée end_i comme action "fin".

• La *concaténation* de $CH_1, ..., CH_n$: $Concat \stackrel{def}{=} (CH_{1[f_1]} \mid CH_{2[f_2]} ... \mid CH_n) \setminus \{end_i, 1 \leq i \leq n-1\}$

où f_i, $1 \leq i \leq n-1$, sont les fonctions de réétiquetage f_i: $begin_{i+1} \rightarrow end_i$. Les actions $begin_1$ et $\overline{end_n}$ sont respectivement les actions "début" et "fin" de l'agent *Concat.*

• *La composition Parallèle* de *n* CHs :

$$Parall \stackrel{def}{=} begin . \overline{begin}^n . Nil \mid CH_{1[f_1]} \mid ... \mid CH_{n[f_n]} \mid end^n . \overline{end} . Nil) \setminus \{begin, end, \{a, b, ...\}\}$$

où f_i, $1 \leq i \leq n$, sont les fonctions de réétiquetage $begin_i \rightarrow begin$, $\overline{end_i} \rightarrow \overline{end}$, et $\{a, b, ...\}$ est l'ensemble des étiquettes d'actions exécutées sur des ports partagés.

• *Le choix non déterministe* entre *n* CHs :

$$Choice \stackrel{def}{=} ((CH_{1[f_1]} + ... + CH_{n[f_n]}) \mid end . \overline{end} . Nil) \setminus \{end\}$$

où f_i, $1 \leq i \leq n$, sont les fonctions de réétiquetage $begin_i \rightarrow begin$ et $\overline{end_i} \rightarrow \overline{end}$.

• La *boucle* sur un CH : $Boucle \stackrel{def}{=} CH \mid (end . Loop + end . \overline{end} . Nil) \setminus \{end\}$

Etant donné que la récursion dans l'agent "Boucle" est bien gardée, son expression CCS dénote un système de transition fini.

2.3 CCS + Contraintes : T^cCCS

Pour tenir compte de la dimension temporelle quantitative des contraintes temporelles nous transférons cette information métrique vers le système de transition dénoté par l'expression CCS décrivant le comportement non temporisé du CH. Le système de transition résultant est représenté par un graphe temporisé appelé *graphe de régions relatives.* Chaque noeud du graphe (appelé région relative) est défini par une expression CCS et un ensemble de relations temporelles entre actions appelé une *zone temporelle relative.* Les notions de zone temporelle et de région ont été définies dans [Dil 89, ACD 90]. Le mécanisme pour générer le modèle (sémantique opérationnelle temporelle) de T^cCCS a été inspiré par les travaux développés dans [BD 91, YSSC 93].

2.3.1. Notations et définitions de base

Les actions visibles d'une expression T^cCCS associé à un chronogramme sont les actions d'entrée, de sortie et les actions "début" et "fin". Pour distinguer les actions internes dues à une interaction entre une paire d'actions auxiliaires et les actions internes dues à une interaction entre une paire d'actions visibles (exécutées sur des ports partagés) nous considérons ces dernières comme actions visibles désignées par actions "internes-visibles". Rappelons que ces actions sont liées par des contraintes temporelles avec d'autres actions. Une action "interne-visible" est étiquetée par τ_a indiquant que l'action τ est le résultat de l'interaction de la paire d'actions $(a, \overline{a})$. L'ensemble des actions de base d'une expression T^cCCS est $Act = N \cup \overline{N} \cup \tau \cup \{\tau_a \mid a \in N\}$, où N et $\overline{N}$ sont des ensembles finis d'étiquettes (d'actions d'entrée) et de coétiquettes (d'actions de sortie). Les éléments de l'ensemble des actions visibles $Act_v = N \cup \overline{N} \cup \{\tau_a \mid a \in N\}$ sont notés l, l', ... et les éléments de Act sont notés α, β, ... Les *actions de garde* sont déterminées par la fonction *garde*: {expressions T^cCCS} $\rightarrow 2^{Act}$, définie comme suit :

$$
\begin{aligned}
&garde\,(Nil) = \varnothing \\
&garde\,(l\,.\,A \setminus l\,) = \varnothing \\
&garde\,(\alpha\,.\,A) = \{\alpha\} \\
&garde\,(A \mid B) = garde\,(A) \cup garde\,(B) \\
&garde\,(A + B) = garde\,(A) \cup garde\,(B)
\end{aligned}
$$

Le terme "$(a\,.\,A \mid \overline{a}\,.\,B\,)\setminus\{a\}$" est équivalent à "$\mu\,.\,(A \mid B)$" où $\mu \in \tau \cup \{\tau_a \mid a \in N\}$.

Une *zone temporelle relative* (ou simplement une *zone*), notée Z, est un polyèdre défini par un ensemble de contraintes liant les occurrences d'un ensemble d'actions désigné par *act(Z)* et appelé l'ensemble d'actions de la zone. Une zone contenant un système d'inéquations n'ayant pas de solution est une *zone inconsistante* (notée $Z^{\varnothing}$) et une zone n'ayant aucune contrainte est une *zone universelle* (notée Z^u). Nous définissons l'*opération d'addition* $\uplus$ sur l'espace Θ des zones temporelles comme suit :

- $Z = Z' \uplus Z''$ contient les contraintes des deux zones Z', Z'' et,
- $act(Z) = act(Z') \cup act(Z'')$

Le uplet $(\uplus, \Theta, Z^u)$ est un *monoïde commutatif.*

Une zone temporelle peut être représentée par un *graphe de contraintes* orienté dont les noeuds représentent les actions de la zone. Une contrainte "$m \leq t_l - t_{l'} \leq M$" liant deux actions l et l' est représentée par deux arcs orientés : de l à l' de poids M et de l' à l de poids $-m$ [Dil 89]. Un graphe de contraintes est dit *fortement connexe* si et seulement s'il existe une contrainte entre chaque paire de noeuds. La *forme canonique* d'un graphe de contraintes d'une zone (noté *cf(Z)*) est un graphe fortement connexe où chaque paire de noeuds est connectée par un arc orienté d'un poids obtenu en résolvant le problème du chemin le plus court entre toutes les paires de noeuds [Tar 83] sur le graphe original.

Nous définissons la fonction *projection* notée $proj : \Theta \times Act \rightarrow \Theta$ telle que pour tout sous-ensemble $Act' \subseteq act(Z)$ de cardinalité supérieure à 2, la zone $proj\,(Z, Act')$ est obtenue en éliminant de *cf(Z)* toutes les contraintes contenant des variables temporelles appartenant au sous-ensemble $act(Z) - Act'$. Nous désignons la zone projetée par $Z_{\Downarrow Act'}$.

L'égalité et l'inclusion de zones sont définies comme suit :

- Etant données deux contraintes $TCons_{l,l'} \equiv (m \leq t_l - t_{l'} \leq M)$ et $TCons'_{l,l'} \equiv (m' \leq t_l - t_{l'} \leq M')$, $TCons_{l,l'} = TCons'_{l,l'}$ *ssi* $m = m'$ et $M = M'$.
- $Z = Z'$ *ssi* $act(Z) = act(Z')$ et $TCons_{l,l'} = TCons'_{l,l'}$, pour toute contrainte $TCons_{l,l'} \in cf(Z)$ et $TCons'_{l,l'} \in cf(Z')$.

- Une zone Z contient une autre zone Z' avec $act\,(Z') \subseteq act\,(Z)$ *ssi* $(Z \uplus Z')_{\Downarrow act(Z')} = Z'$; on dira que l'information quantitative de Z est non pertinente à Z'.

En vue de décrire les règles temporelles de T^cCCS, nous donnons dans ce qui suit les définitions des zones d'*environnement*, "*First*" et *Future.*

Définition 2.4. La *zone d'environnement* (notée *Env*) contient toutes les contraintes d'un opérateur "conjonctif". Rappelons que les contraintes d'assertion liant des actions "internes-visibles" sont des contraintes que le comportement d'un chronogramme doit satisfaire.

Définition 2.5. La *zone Future* d'une action visible l (notée $Futur_l$) est composée de toutes les contraintes de réaction pour lesquelles l'action l est une action source.

Définition 2.6. Soit $l \in garde\,(A)$. La *zone "First"* de l'action l vis-à-vis de $Act' \subseteq atv(A)$ définit l'ordre d'exécution entre l'action l et les actions de Act' :

$$First^{l}_{Act'} \stackrel{def}{=} \{t_l \leq t_{l'} \mid l' \in Act'\}$$

Nous associons une horloge Clk_l à toute action $l \in Act_v$. L'horloge d'une action mesure le temps pour déterminer le temps de son exécution conformément aux contraintes temporelles relatives à l'action. L'horloge associée à un élément de l'ensemble d'actions d'une zone d'environnement est initialisée à 0 aussitôt que la première action de cet ensemble est exécutée. L'horloge d'une action de sortie qui n'est pas liée par une contrainte d'assertion est initialisée à 0 dès que sa source est exécutée. L'horloge d'une action τ_a est initialisée à 0 aussitôt que la source de $\overline{a}$ est exécutée.

L'occurrence d'une action dans une expression T^cCCS est *active ssi* son horloge associée a été initialisée. Il est aisé de vérifier que $garde\,(A) \subseteq atv(A)$. Nous désignons par $atv\,(A)$ l'ensemble des actions actives de A et par $atv^o\,(A)$ le sous-ensemble de $atv\,(A)$ contenant les actions de sortie et les actions "internes-visibles" qui ne sont pas des actions de garde.

2.3.2. Graphe de régions relatives

Un terme T^cCCS décrivant un chronogramme est défini d'une part, par l'expression CCS spécifiant l'ordre partiel entre le actions du chronogramme et d'autre part, par une zone temporelle relative Z. Lorsqu'une action de garde est exécutée, de nouvelles contraintes sont ajoutées à la zone temporelle précédente. Comme nous préservons la sémantique d'entrelacement de CCS (les actions sont totalement ordonnées), seules les contraintes relatives à une action sont ajoutées à la fois. Le couple (expression CCS, zone temporelle relative) est une *région relative* notée r.

Définition 2.7. Soit A_0 l'expression CCS décrivant un chronogramme et dénotant le système de transition $(S, Act, \rightarrow, A_0)$. Soit $\mathcal{Z}$ l'ensemble des zones temporelles relatives définies par les contraintes temporelles entre les actions de l'ensemble Act. Un *graphe de régions relatives* est caractérisé par le uplet $(R, \twoheadrightarrow, r_{init})$ où :

- R est l'ensemble des régions $\{A_i|_{Z_i} \mid i=0, 1, \ldots\}$ où $A_i \in S$ est un terme CCS et $Z_i \in \mathcal{Z}$ est une zone.
- $\twoheadrightarrow \subseteq R \times R$ est la relation de transition. On distingue deux types de transitions : la progression du temps à l'intérieur de la région où les horloges des actions actives mesurent le temps à la même cadence (balayage uniforme de la zone) et des transitions entre régions engendrées par l'exécution d'une action.
- r_{init} est la région initiale définie par l'agent décrivant le chronogramme avant l'exécution de l'action "début" et la contrainte temporelle : $0 \leq t_{begin} - t_{CH} \leq 0$, où t_{begin} est le temps d'occurrence de l'action "début" et t_{CH} est une origine virtuelle du temps.

2.3.3. Sémantique opérationnelle

Nous décrivons à présent les règles temporelles de T^cCCS permettant d'associer à un terme décrivant un chronogramme un graphe de régions relatives.

Définition 2.8. Afin de distinguer le temps d'occurrence des actions exécutées dans le passé (produisant des transitions entre régions) et le temps d'occurrence des actions futures, nous introduisons les notions de variable temporelle du passé et de variable temporelle du futur [YSSC 93] notées respectivement $\cdot t$ et $t\cdot$. Les sous-ensembles d'actions d'une zone temporelle représentées par leurs variables temporelles du passé et du futur sont notées respectivement $\cdot act(Z)$ et $act^{\cdot}(Z)$.

Les variables temporelles du passé et du futur on été introduites pour générer un graphe de régions dans lequel les zones temporelles ont un minimum de variables temporelles. Ce qui permet de réduire les coûts en temps de calcul et en mémoire lors de la génération du graphe en éliminant des zones temporelles les actions représentées par des variables temporelles du passé qui ne sont pas nécessaire pour des évaluations futures. En effet, le but de générer un graphe de régions est de vérifier les contraintes de propriété (Section 2.1). Donc, les variables temporelles à éliminer sont celles des actions qui ont été exécutées et qui ne sont pas liées par des contraintes de propriétés $(\{l | l \notin Prop\})$.

Si un chronogramme contient une boucle sur un CH_i, le CH_i doit vérifier la contrainte $^{\cdot}t_{\tau_{end}} - {}^{\cdot}t_{begin_{CH_i}} > 0$, où τ_{end} est l'action issue de l'interaction de la paire d'actions $(end_{CH_i}, \overline{end}_{CH_i})$. Ceci permet de détecter des exécutions infinies sans progression du temps. Cette condition est désignée dans la littérature par l'"*exigence non-Zeno*" [AL 91].

Nous donnons dans ce qui suit, "à la Plotkine", les règles décrivant l'évolution temporelle de $A|_Z$:

- *Règle* 1. (*Blocage*) Aucune action ne peut être exécutée si la zone temporelle de la région est inconsistante:

$$\frac{Z = Z^{\emptyset}}{A|_Z \overset{\alpha}{\nrightarrow}}$$

- *Règle* 2. (*Restriction d'urgence*) Une action $l \in garde(A)$ ne peut pas être exécutée s'il existe d'autres actions dont l'exécution est plus urgente. Ceci ce traduit par le fait que la zone $First^{l}_{guard(A)}$ est incohérente avec la zone Z :

$$\frac{First^{l}_{guard(A)} \uplus Z = Z^{\emptyset}}{A|_Z \overset{\alpha}{\nrightarrow}}$$

- *Règle* 3. (*Mouvement silencieux*) Les actions internes de l'ensemble *garde* (*A*) doivent être exécutées avant toute autre action visible. L'exécution d'une action interne donne lieu à un mouvement silencieux vers une région ayant la même zone temporelle que celle de la région source. En effet, les zones temporelles contiennent des contraintes temporelles liant uniquement l'occurrence des actions visibles :

$$\frac{Z \neq Z^{\emptyset}, \tau \in garde(A)}{A|_Z \overset{\tau}{\rightarrow} A'|_Z}$$

• *Règle* 4. (*Restriction d'environnement*) : Soient $l \in garde(A)$ et Env_l sa zone d'environnement. Si la zone "First" de l'action l vis-à-vis des actions de $act(Env_l)$ qui n'ont pas été exécutées est incohérente avec Env_l, alors l ne peut pas être exécutée. Cette règle impose un ordre d'exécution dicté par les contraintes d'assertion :

$$\frac{First^l_{Act(V)} \uplus Env_l = Z^{\varnothing}}{A|_Z \overset{l}{\not\rightarrow}} \quad \left(V = \begin{cases} Env_l & si\ act^{\bullet}(Z) \cap act(Env_l) = \varnothing \\ Env_{l \Downarrow (act^{\bullet}(Z) \cap act(Env_l))} & autrement \end{cases} \right)$$

• *Règle* 5. (*Mouvement entre régions*) Une action visible $l \in garde(A)$ peut être exécutée *ssi* (1) les conditions des règles 1, 2 et 4 ne sont pas satisfaites et (2) l'ensemble *garde* (A) ne contient aucune action interne :

$$\frac{Z \neq Z^{\varnothing}, First^l_{guard(A)} \uplus First_{act(Env)} \uplus Z \neq Z^{\varnothing}, \tau \notin garde(A)}{A|_Z \overset{l}{\rightarrow} A'|_{Z'}}$$

Lorsque l'action l est exécutée, l'expression A évolue vers l'expression A' selon les règles d'action de CCS et une nouvelle zone Z' obtenue comme suit :

(1) La zone *Future* et la zone d'*environnement* (Env_l) de l'action l ainsi que sa zone "*First*" vis-à-vis de "$act^{\bullet}(Z \uplus Env_l) - atv^{o}(A)$" sont additionnées à la zone d'origine Z :

$$Z''' = Z \uplus Futur_l \uplus Env_l \uplus First^l_{act^{\bullet}(Z \uplus Env_l) - atv^{o}(A)}$$

Notons que la zone d'environnement Env_l d'une action d'entrée l est additionnée à la nouvelle zone *ssi* Env_l n'a pas été incluse auparavant :

$$Env_l = \begin{cases} Env & if\ l \notin act(Z)\quad et\ l \in act(Env) \\ Z^u & if\ l \in act^{\bullet}(Z) \end{cases}$$

D'autre part, l'ordre d'exécution entre l'action l et les éléments de $atv^{o}(A)$ ne doit en aucun cas restreindre la zone temporelle d'origine, sinon le chronogramme est inconsistant (voir section 4 pour plus d'informations).

(2) La variable temporelle du futur de l'action l est substituée à sa variable temporelle du passé :

$$Z'' = Z''' / [t_l^{\bullet} \leftarrow {}^{\bullet}t_l]$$

(3) Réduction de la taille de la zone : Si l dénote une action "interne-visible" issue de l'interaction (end_{leaf_i}, $\overline{end_{leaf_i}}$), les variables temporelles du passé de toutes les actions du CF i sont éliminées de $cf(Z'')$ pour éviter la génération d'un graphe infini. Si $l \neq \tau_{end_{leaf_i}}$ et l n'est pas liée par une contrainte de propriété ($l \notin Prop$), sa variable temporelle est éliminée de $cf(Z'')$. La zone Z' est définie comme suit :

$$Z' = \begin{cases} Z''_{\Downarrow (act(Z'') - \{l\}) \cup (\{l\} \cap Prop)} & si\ l \neq \tau_{end_{leaf_i}} \\ Z''_{\Downarrow (act(Z'') - (\{l\} \cup Prop_{leaf_i})} & si\ l = \tau_{end_{leaf_i}} \end{cases}$$

2.3.4. Relation d'équivalence

Définition 2.9. Les régions $r_A = A|_{Z_A}$ et $r_B = B|_{Z_B}$, r_A, $r_B \in \mathcal{R}$, sont équivalentes ($r_A \cong r_B$) *ssi* $Z_A = Z_B$ et $\forall$ $l \in Act_v$:

$$(i).\quad r_A \xrightarrow{l} r_{A'} \quad \Rightarrow \quad \exists r_{B'},\, r_B \xrightarrow{l} r_{B'} \quad et \quad r_{A'} \cong r_{B'}$$

$$(ii).\quad r_B \xrightarrow{l} r_{B'} \quad \Rightarrow \quad \exists r_{A'},\, r_A \xrightarrow{l} r_{A'} \quad et \quad r_{B'} \cong r_{A'}$$

Proposition 3.1. Le quotient $\mathcal{R} / \cong$ est fini.

Cette proposition peut être prouvée en se basant sur la méthode de construction du quotient fini associé à un automate temporisé présentée dans [ACD 90, BES 92]. Un graphe de régions relatives est une partition de ce "quotient fini" par rapport à la relation d'équivalence $\cong$.

3 Vérification de la consistance d'un CH

La spécification d'un chronogramme peut être inconsistante si certaines contraintes d'assertion et/ou de réaction ne sont pas satisfaites par les comportements du chronogramme. De plus, le concepteur peut spécifier un ensemble de contraintes de propriété que le chronogramme doit satisfaire durant tout comportement. Dans cette section nous utilisons le graphe de régions relatives pour résoudre ces problèmes de vérification.

3.1 L'endoconsistance

La vérification de l'endoconsistance (ou consistance interne) d'un CH consiste à s'assurer que toute action de sortie ou "interne-visible" peut être exécutée à l'intérieur de l'intervalle temporelle défini par la contrainte la liant à sa source.

Définition 3.1. La spécification d'un CH contient un *blocage* (en anglais, *deadlock*) si et seulement s'il n'existe aucun instant temporelle pour lequel une action active peut être exécutée. En d'autres termes, un CH contient un *blocage* si et seulement s'il existe une région $A|_Z$, $A \neq Nil$, dans son graphe n'ayant aucun successeur.

Théorème 4.1. Un CH est endoconsistant *ssi* :

(*i*) il ne contient pas de *blocage* et,

(*ii*) toute région $A|_Z$ dans son graphe satisfait les relations :

$$a.\,(Z \uplus First^{l}_{atv^o(A)})\Downarrow_{atv^o(A)\,\cup\,\{l\}} = Z\Downarrow_{atv^o(A)\,\cup\,\{l\}}$$

$$b.\,(Z \uplus Futur_l)\Downarrow_{act(Futur)} = Futur_l$$

pour tout $l \in garde\,(A)$ satisfaisant les conditions des règles 2 et 4.

Preuve : La condition (*i*) est évidente.

(*ii*) *a.* Etant donné que les actions de $atv^o(A)$ ne peuvent pas être exécutées ($atv^o(A) \cap garde\,(A) = \varnothing$), l'information temporelle de la zone $First^{l}_{atv^o(A)}$, indiquant que l'action l est exécutée avant toute action appartenant à $atv^o(A)$, doit être non pertinente à Z. Sinon, la liberté d'occurrence de certaines actions de $atv^o(A)$ est restreinte par l'occurrence de l.

(*ii*) *b.* Etant donné que les actions de sortie peuvent être liées par des contraintes d'assertion avec des contraintes d'entrée, l'ensemble d'actions d'une zone peut contenir une action de sortie ou "interne-visible" dont la présence est due à l'inclusion de la zone d'environnement d'une action exécutée dans le passé. Pour vérifier que le champ d'occurrence des actions de sortie ou "internes-

visibles" (défini par les contraintes de réaction) n'est pas restreint par les contraintes d'assertion, il suffit de s'assurer que la zone Z d'une région contient les zone futures des ses actions de garde. ❑

3.2 L'extraconsistance

L'extraconsistance d'un CH (ou consistance externe) est vérifiée en s'assurant que chaque région accessible est cohérente avec les contraintes de propriété.

Théorème 4.2. Un chronogramme endoconsistant est extraconsistant *ssi* la zone Z de chacune de ses régions accessibles satisfait la propriété suivante :

$$\forall prop_{\{l,\, l'\}} \in \{prop_{\{l,\, l'\}} \mid \{l, l'\} \subseteq {}^{\bullet}act(Z)\},\ (Z \uplus req_{\{l,\, l'\}})_{\Downarrow \{l,\, l'\}} = Z_{\Downarrow \{l,\, l'\}}$$

Preuve : Une région avec la zone temporelle Z satisfait la contrainte de propriété $prop_{\{l,\, l'\}}$, $\{l, l'\} \subseteq {}^{\bullet}act\ (Z)$ *ssi* l'information temporelle de la contrainte $prop_{\{l,\, l'\}}$ est non pertinente à Z. En d'autres termes, la contrainte $prop_{\{l,\, l'\}}$ doit contenir la zone $Z_{\Downarrow \{l,\, l'\}}$. ❑

3.3 Vérification

La vérification de la consistance d'un chronogramme (interne et externe) revient à énumérer les régions de son graphe sans le construire. Le parcours du graphe consiste à simuler exhaustivement le comportement du chronogramme à partir de sa région initiale. La vérification et la génération du graphe sont réalisées simultanément (vérification "à la volée"). La vérification est arrêtée lorsque toutes les régions accessibles ont été énumérées ou lorsqu'une inconsistance est détectée.

La figure 3 présente un algorithme énumératif pour l'exploration du graphe de régions dénoté par le chronogramme. La fonction "*Générer_successeurs*" retourne les successeurs possibles d'une région selon les règles de T[c]CCS. La fonction *Consistance* retourne *vrai ssi* (1) toute action active de sortie ou "interne-visible" peut être exécutée à n'importe quel instant défini par la zone temporelle de la région (Théorème 4.1) et (2) la région satisfait les contraintes de propriété.

La stratégie adoptée pour parcourir le graphe (l'exploration "à la volée") permet la détection des régions futures avec des zones inconsistantes en vérifiant la condition (*ii*) du théorème 4.1.

```
1. Initialisation.
    Région_initiale = ( expression_CCS, zone_initiale);
    Régions_accessibles = {région_initiale};
    Régions_visitées = ∅;
2. Boucle.
    Tant que Régions_accessibles ≠ ∅ Fait
    Choisir Région = (A, Z) de Régions_accessibles;
    Régions_visitées = Régions_visitées ∪ {Région};
    Régions_accessibles = Régions_accessibles – {Région};
    Si Consistance {(A, Z)} = faux alors retourner ("CH inconsistant");
    Sinon     Successeurs = Générer_successeurs {Région};
              Si (Successeurs = ∅ and A ≠ Nil ) alors retourner ("Blocage!");
              Sinon Régions_accessibles=Régions_accessibles ∪ Successeurs - Régions_visitées;
    Findefait
    retouner ("CH consistant")
```

Figure 1. L'algorithme de vérification

4 Conclusion et perspectives

Nous avons présenté dans cet article une approche formelle de vérification des chronogrammes hiérarchiques (CHs). Tout d'abord, nous avons montré que CCS a le pouvoir d'expression pour spécifier les ordres partiels entre les actions engendrés par les contraintes temporelles. Nous avons ensuite montré comment transférer le temps métrique des contraintes temporelles vers le système de transition dénoté par l'expression CCS décrivant le CH. Le modèle résultant est un *graphe de régions relatives* ; chaque région du graphe est caractérisée par le couple (*expression CCS, un ensemble de contraintes temporelles*). La vérification de la consistance d'un CH consiste à parcourir son graphe de régions sans le construire.

En vue d'évaluer les performances de la méthode de vérification, nous sommes en train d'implémenter un prototype opérant sur des données fournies par un éditeur de chronogrammes permettant la saisie des spécifications.

Il a été souligné dans [KC 94], après plusieurs expériences de spécification de systèmes réels, l'utilité d'enrichir les opérateurs de composition avec deux nouveaux opérateurs : "*choix déterministe retardé*" et "*gestion d'interruptions*". D'autre part, pour spécifier aisément le comportement de certains systèmes au plus niveau d'abstraction tels que les circuits "pipeline", il serait intéressant d'introduire des contraintes hiérarchiques. Nous projetons dans un travail futur d'étendre l'algèbre T^cCCS pour comprendre ces extensions.

Concernant la méthode de vérification et dans l'optique d'obtenir des résultats en temps raisonnable, il serait intéressant d'explorer les méthodes basées sur les ordres partiels pour limiter le problème d'explosion d'états dû à la sémantique d'entrelacement de CCS. La génération du modèle pourrait être dirigée par les contraintes de propriété (les contraintes à vérifier).

Finalement, la méthode de vérification étudiée dans cet article peut comprendre la vérification des *propriétés de sûreté* relatives aux comportements logiques (qualitatifs) des systèmes matériels numériques. En effet, il a été montré dans la littérature (voir par exemple : [Ber 92, Ant 93, TB 93]) que cette classe de propriétés définit des langages qui peuvent être décrits par des opérateurs d'observation. Ces opérateurs peuvent être spécifiés par des chronogrammes communiquant via la composition parallèle avec le chronogramme hiérarchique décrivant le comportement de l'interface matérielle.

Bibliographie

[AB 92] T. Amon, G. Borriello. *An Approach for Symbolic Timing Verification*. In DAC Proc., June 1992.

[AL 91] M. Abadi, L. Lamport. *An Old-fashioned Recipe for Real-Time*. In Real Time: Theory and Practice, LNCS 600, 1991.

[ACD 90] R. Alur. C. Courcoubetis, D. Dill. *Model Checking for Real-Time Systems*. In Proceeding of the 17th International Colloquium on Automata Languages and Programming, Spring-Verlag, LNCS 443, 1990.

[AHBB 93] T. Amon, H. Hulgaard, S. Burns, G. Borriello. *An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems*. In ICCD Proc., October 1993.

[Ant 93] C. Antoine. *CHLOE: un Editeur Compilateur de Chronogrammes Logiques*. Thèse, Université de Provence (Marseille), France, Novembre 1993.

[BD 91] B. Berthomieu, M. Diaz. *Modeling and Verification of Time Dependent Systems Using Time Petri Nets*. In IEEE Transaction on Software Engineering, Vol. 17, No. 3, March 1991.

[Ber 92] B. Berkane. *Vérification des Systèmes Matériels Numériques Séquentiels Synchrones*. Thèse, Institut National Polytechnique de Grenoble, France, Octobre 1992.

[BES 93] A. Bouajjani, R. Echahed, J. Sifakis. *On Model Checking for Real-Time Properties with Durations*. In Proceeding of the 8th Annual IEEE Sym. on Logic in Computer Science, Montreal, Canada, June 1993.

[BC 94] B. Berkane, E. Cerny. *Modeling and Verifying Timing Diagrams using Time-constrained CCS*. Rapport technique, IRO/Université de Montréal, Janvier 1994.

[Dil 89] D. Dill. *Timing Assumptions and Verification of Finite-State Concurrent Systems*. In Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, Lecture Notes in Computer Science 407, Spring-Verlag, 1989.

[DM 92] K. McMillan, D. Dill. *Algorithms for Interface Timing Verification*. In Proceeding of the 2nd ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, 1992.

[HNSY 92] T. Henzinger. X. Nicollin, J. Sifakis, S. Yovine. *Symbolic Model Checking for Real-Time Systems*. In Proceeding of the 7th IEEE Symposium on Logic in Computer Science, 1992.

[KDC+ 93] K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine, A. Silburt. *Integrating Behavior and Timing in Executable Specifications*. In Proc. of the IFIP CHDL-93 Conf., Otawa, Canada, April 1993.

[KC 93] K. Khordoc, E. Cerny. *Specifying Systems with Action Diagrams*. Rapport technique, IRO/Université de Montréal, Avril 1993.

[KC 94] K. Khordoc, E. Cerny. *Modeling Cell Processing Hardware with Action Diagrams*. To appear in Proc. of ISCAS'94.

[Lew 89] H. R. Lewis. *Finite-State Analysis of Asynchronous Circuits with Bounded Temporal Uncertainty*. Technical Report TR-15-89, Havard University, 1989.

[Mil 89] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science 89.

[NS 91] X. Nicollin, J. Sifakis. *An Overview and Synthesis on Timed Process Algebras*. In Proceeding of the 3rd Workshop on Computer Aided Verification, Aalboug, Denmark, July 1991.

[Tar 83] R. E. Tarjan. *Data Structures and Networks Algorithms*. SIAM, Philadelphia, 1983.

[TB 93] G. Thuau, B. Berkane. *A Unified Framework for Describing and Verifying Hardware Sequential Systems*. In Journal of Formal Methods in System Design, Volume 2, Number 3, july 1993.

[YSSC 93] T. Yoneda, A. Shibayama, B.-H. Schlingloff, E. Clark. *Efficient Verification of Parallel Real-Time Systems*. In Proc. of the 5th Workshop on Computer Aided Verification, Elounda, Greece, June 1993.