I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1402

A FRAMEWORK TO ANALYSE SYNCHRONOUS DATA-FLOW SPECIFICATIONS

MIRABELLE NEBUT , SOPHIE PINCHINAT

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

A Framework to Analyse Synchronous Data-Flow Specifications

Mirabelle Nebut , Sophie Pinchinat*

Thème 1 — Réseaux et systèmes
Projets Espresso et S4

Publication interne n1402 — Novembre 2001 — 59 pages

Abstract: Presence and absence of signals inside a reaction are inherent to the synchronous paradigm, as well as clocks which are sets of instants that indicate when a given condition is fulfilled over a sequence of reactions (e.g. when a signal is present). Clocks are essential to capture the control in data-flow specifications; more generally relations between clocks should be analyzed to verify some properties, e.g. to detect inconsistencies in specifications. These relations express particular safety properties many of which can be verified without considering the dynamic of systems, by means of a static abstraction. We propose a language $\mathcal{CL}$ to describe such properties and prove it decidable. Model-checking is derived for SIGNAL programs, on the basis of a translation from the static abstraction of SIGNAL into $\mathcal{CL}$. Links with existing models and abstractions for the analysis of SIGNAL programs are largely discussed.

Key-words: synchronous paradigm, data-flow language, clock, decision procedure, abstraction, safety property, model-checking, SIGNAL

(Résumé : tsvp)

Thanks are due to Paul Le Guernic for extensive comments about sections 2, 3 and 5 and useful discussions.

* {mnebut,pinchina}@irisa.fr

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Un cadre pour l'analyse de spécifications flot de données synchrones

Résumé : La présence et l'absence des signaux à l'intérieur d'une réaction sont inhérentes au paradigme synchrone, de même que les horloges : ces ensembles d'instants indiquent quand une condition donnée est vérifiée au cours d'une suite de réactions (par exemple quand un signal est présent). Les horloges sont essentielles à la description du contrôle des spécifications flot de données, mais d'une manière plus générale l'analyse des relations entre horloges permet de vérifier certaines propriétés, par exemple de détecter des incohérences dans les spécifications. Ces relations expriment des propriétés de sûreté particulières dont une grande partie peut être vérifiée sans considérer la dynamique des systèmes, au moyen d'une abstraction statique. On propose le langage $\mathcal{CL}$ pour décrire de telles propriétés. Une procédure de décision est donnée pour $\mathcal{CL}$, de laquelle est dérivé le model-checking de programmes SIGNAL grâce à une traduction de l'abstraction statique de SIGNAL en $\mathcal{CL}$. Le lien entre ce travail et les modèles et analyses existants pour SIGNAL est largement développé.

Mots clés : paradigme synchrone, langage flot de données, horloge, procédure de décision, abstraction, propriété de sûreté, model-checking, SIGNAL

Table of Contents

1 Introduction

Synchronous languages [26, 45, 28, 10] have been proposed to specify reactive systems, which interact continuously with the environment they are connected to. The imperative, state-based and data-flow programming paradigms both involve the notion of logical instants and presence/absence of signals. It is particularly apparent in imperative languages like ESTEREL [10], well adapted to problems for which the control-handling aspects are prevalent. A signal of type pure event is only an "impulse": the information it carries is limited to its *status*, that is its presence or absence (valued signals also exist). Basic instructions specify that the system broadcasts signals (`emit S`) or reacts to the presence or absence of signals (e.g. `present S then` p `else` q). The equational data-flow languages like SIGNAL [45] and LUSTRE [28] are better adapted to problems where data-flow is prevalent. Specifications are systems of equations which describe the possible values carried by signals along the time. At a given instant, a signal (e.g. of type boolean, integer, etc) can be absent or present, hence carrying a significant value. The set of instants at which it is present is called its *clock*.

In the synchronous paradigm models of executions of the system are sequences of instantaneous reactions along a logical time line, which attach a status to each signal (and possibly a value, accordingly). Such an assignment is a *valuation*. For data-flow programs, these time-indexed sequences of valuations are generally called *flows*, or traces. A program is then associated a particular set of possible flows, called a *process* here.

Contrary to imperative languages, data-flow specifications do not make the control explicit: hence it must be synthesized. The powerful notion of *clocks* is essential for this purpose since clocks are *sets of instants* that characterize the instants *when* given conditions hold. The clock of a signal is particularly important: it indicates when the signal carries a significant value which can then be involved in computations. But clocks of interest can also be the set of instants when the sum of the values of two signals is positive, or more generally when an input has to be read, when a particular computation has to be performed, etc.

Clocks are handled differently in LUSTRE and SIGNAL. In LUSTRE a basic clock is attached to the program, from which "slower" clocks are *functionally* extracted by means of boolean flows: the new sequence is composed of instants at which the flow carries the value *true*. More generally in SIGNAL equations state *relations* between clocks, among which the extraction principle of LUSTRE. Then in the case of LUSTRE it is quite easy to derive an order for computations which meets dependencies between variables, while the larger SIGNAL expressiveness makes necessary a deduction mechanism and a real synthesis. Nevertheless in both cases and even in the whole synchronous paradigm, the analysis of relations between clocks allows to obtain code of higher quality and e.g. to detect inconsistencies. Therefore powerful techniques to automate deductions and more generally perform verifications need to be developed.

Relations between clocks express *instantaneous* properties, which describe only what happens inside a reaction, with no reference to the preceding or following ones. E.g. "the level will never exceed 10" (for all instant t, *level* at t is lower than 10), or "the alarm is never raised" (for no instant t, *alarm* is present), or "each time the tank becomes empty, then the faucet is turned on" (for all instant t, if *level* at t is equal to 0 then *faucet* is present) are instantaneous properties. They fall into the class of *safety* properties, which are the most important

ones concerning reactive systems. On the contrary, and as expected, liveness properties (e.g. "there exists an instant when the train is stopped") or properties that involve references to more than a single instant such as "the value of x increases from one instant to the next one" are not instantaneous properties.

Standard approaches to establish safety properties rely on operational models which take account of the dynamic of systems. It is also the case for instantaneous properties, e.g. the fact that "the level will never exceed 10" likely strongly depends on the evolution of the signal *level* along the time. Nevertheless many of them can be verified while abstracting from dynamic aspects (e.g. "the emptiness of the tank turns on the faucet"), but instead focusing on static features (obtained by a *static abstraction* of programs).

The present work proposes a language called $\mathcal{CL}$ (read "clock language") to express instantaneous properties of processes (that hold in all possible flow of the process). Basically, the formulas of the language (presented in Sect. 6) express inclusions between clocks inside flows. Provided clocks are defined using a decidable theory, the language $\mathcal{CL}$ is proved decidable and a decision procedure is given in Sect. 7. Moreover, Sect. 8 shows in the case of SIGNAL that the static abstraction of data-flow specifications translates naturally in $\mathcal{CL}$: model-checking can then be easily derived from the decision algorithm. In this context the $\mathcal{CL}$ language as well as its decision procedure strictly generalize existing works on SIGNAL, where clocks are defined using only a boolean theory, see e.g. [1]).

A large part of the document is first dedicated to the presentation of existing works applied to SIGNAL and provided by the environment POLYCHRONY. In Sect. 2 are given two classical semantics for data-flow processes, useful to present the SIGNAL language as well as two related analyses in Sect. 3. A flow semantics is first presented and used to describe particular classes of processes: deterministic, endochronous (usually characterized by an operational semantics, see e.g. [57]) and static (so far informally defined). Then a semantics of processes as transition systems is given and compared with the flow semantics. In Sect. 4 are recalled a few principles about abstraction-based analyses, in particular the framework of abstract interpretation, in order to present in Sect. 5 abstractions (and related analyses) used in POLYCHRONY.

These analyses are very diverse: e.g. authors of the work [11] claim its relation with abstract interpretation, while other abstractions are described by means of syntactical transformations of SIGNAL programs; some analyses address dynamic features while others are dedicated to the verification of instantaneous properties over the static part of processes. Moreover the particularly important abstraction by synchronizations is quite original in its principle, though it is only a particular case of boolean abstraction. For these reasons, the whole set of studies involving abstractions in POLYCHRONY can only gain clarity from a general survey which would establish classifications and comparisons both between classes and with classical abstractions.

We propose in this document a classification into on the one hand what we call *structural abstractions* and in the other hand semantical abstractions into data-domains. Structural abstractions are emphasized: they decompose a program into the parallel composition of two sub-programs, one of which is kept as an abstraction of the original one. The link with abstract interpretation is highlighted and a few examples picked up from the POLYCHRONY environment are enumerated; in particular we present how some abstract processes involving only boolean features are described by means of relations between particular clocks, in the "Clock Algebra". It gives rise to a discussion about analyses of *admissible valuations* of a system (admissible valuations contain the information kept by the static abstraction): it is shown that in the general case any analysis of the trajectories of a system necessitates an analysis of its admissible valuations.

When dealing with abstractions into data domains, it is pointed that one cannot do without determining first the status of values before handling their values. Therefore clocks should practically be used as objects of analyses even in the case of non-boolean systems, provided that values of variables are taken into account. Examples motivate an extension of the Clock Algebra, which leads to the definition of the $\mathcal{CL}$ language in Sect. 6.

2 The Synchronous Data-Flow Paradigm

Data-flow synchronous languages (such as SIGNAL [45, 25, 9] and LUSTRE [16]) specify a system by a set of equations. An equation describes a set of data-flows, i.e. sequences of flows of data carried by variables called signals. According to [7] the data-flow formalism is well adapted to problems where data-flow is prevalent, while state-based formalisms (like STATECHARTS [33]) or imperative languages (like ESTEREL [10]) are well adapted to problems where control-flow is prevalent.

The content and notations of this section stand for the whole data-flow paradigm, even if the presentation is particularly adapted to SIGNAL. Notations are first given in Sect. 2.1. A trace and flow semantics is then described: valuations, traces, flows and clocks are presented in Sect. 2.2. Then processes are presented in

Sect. 2.3. Standard classes of processes (*deterministic*, *endochronous* and *static*) are presented by means of the flow semantics in Sect. 2.4. Finally semantics of data-flow specifications as transition systems is presented in Sect. 2.5.

2.1 Notations

A data-flow specification describes a system by a set of *equations* involving typed *variables* (e.g. boolean, integer, etc) named *signals*[1]. Informally a signal x is a *data-flow* : a sequence of values $(x_t)_{t\in\mathbb{T}}$, where $\mathbb{T}$ is a set of instants called the *time referential*. To simplify, we choose $\mathbb{T}$ as a totally ordered and countable set[2], say $\mathbb{N}$. S is a set of variables whose values have domain D. $\star$ is a special value[3] (it should be read as "absence" or "bottom") such that $\star \notin D$, and $D^\star$ stands for $D \cup \{\star\}$. At a given instant $t \in \mathbb{T}$, a variable x can be *absent* (denoted by $x_t = \star$) or *present*: $x_t = v$ for some value $v \in D$. The information of the presence or absence of a variable at a given instant is called its *status*. It corresponds to the special data type of *pure events* $\{true, \star\}$ (present with value *true* or absent). The *clock* of the variable x, written $\widehat{x}$, is the set of its instants of presence ($\widehat{x} \subseteq \mathbb{T}$). E.g. on the example given page 11 on Fig. 4, $\{t_3^a, t_4^a\} \subseteq \widehat{b}$ but $t_2^a \notin \widehat{b}$. Variables which have the same clock are said to be *synchronous* (they are both absent or present at the same time).

2.2 Traces and Flows as Models for Specifications

Data-flow languages are naturally given a trace semantics: a trace is a sequence of values carried by variables. Specifications are then interpreted by a set of traces. A simple trace semantics partly taken up from works about SIGNAL is presented here. Other presentations can be found in [53, 8, 9]. Events and valuations are first presented in Sect. 2.2.1 as components of traces (Sect. 2.2.2). Then flows are defined in Sect. 2.2.3 as particular traces. Finally the clock of variables is defined with respect to the trace semantics in Sect. 2.2.4.

2.2.1 Events and Valuations

A *valuation* on a subset A of S (or simply valuation) is a function which assigns a value in $D^\star$ to each variable in A. It is traditionally called an *event* (not to be confused with the type of pure events) but we prefer this designation in accordance with notations of Sect. 6. $\#$ is the special *blocking* valuation which indicates an inconsistency. Formally:

Definition 2.1 (Valuation, Event) *A valuation on A is a function V: $A \to D^\star$ (whose set is denoted by $\mathcal{V}_A$) or $\#$. $\mathcal{V}_A^\#$ stands for $\mathcal{V}_A \cup \{\#\}$, with typical elements V, V_1, V_2, etc.*

The only valuation on the empty set of variables is denoted by $1_\mathcal{V}$ ($\mathcal{V}_\emptyset$ is the singleton $\{1_\mathcal{V}\}$). The valuation which maps every variable of A to $\star$ is the *silent* valuation ${}^\star V_A$. A non-silent valuation is said *significant* or *concrete*. The *restriction* of a valuation $V \in \mathcal{V}_A^\#$ to $A_1 \subseteq A$ is denoted by $V_{|A_1}$.

The *synchronous product* combines the valuations $V_1 \in \mathcal{V}_{A_1}^\#$ and $V_2 \in \mathcal{V}_{A_2}^\#$ in such a way that: if V_1 and V_2 agree on the values of variables in $A_1 \cap A_2$ then their composition gives $V_1 \cup V_2$, else the blocking valuation.

Definition 2.2 (Composition of valuations) *The synchronous product of valuations $\cdot$ is defined by:*

$$\begin{array}{rccl} \cdot : & \mathcal{V}_{A_1}^\# \times \mathcal{V}_{A_2}^\# & \to & \mathcal{V}_{A_1\cup A_2}^\# \\ & (V_1, V_2) & \mapsto & \left\{ \begin{array}{l} \# \textbf{ if } V_1 = \# \textit{ or } V_2 = \# \textbf{ else} \\ V_1 \cup V_2 \textbf{ if } \forall x \in A_1 \cap A_2, V_1(x) = V_2(x) \textbf{ else} \\ \# \end{array} \right. \end{array}$$

For a given set of variables A, $(\mathcal{V}_A^\#, \cdot, 1_\mathcal{V})$ is a commutative and idempotent monoid (a set endowed with an associative internal law), whose absorbent element is $\#$.

[1] Variables should rigorously be named *ports*, while the term *signal* refers to the set of values carried by a port. We abuse these terms and generally use the term *variable* or *signal variable* in the following.

[2] [8] and more recently [44] show that a partial order is sufficient, provided that the subset of instants denoted by a signal is totally ordered (then the previous value of a signal has a meaning).

[3] Absence is traditionally denoted by $\perp$, not to be confused with the minimal element of a lattice.

2.2.2 Traces

A trace is a model intended to represent an execution of the system. Since the system is not supposed to terminate, it is an *infinite*[4] succession of valuations over the time referential $\mathbb{N}$. Formally:

Definition 2.3 (Trace) *The set of traces on A is denoted by $\mathcal{T}_A^\#$, with typical elements T, T_1, T_2. A trace in $\mathcal{T}_A^\#$ is a function $T: \mathbb{N} \to \mathcal{V}_A^\#$.*

A trace which maps no instant on # is said to be *non-blocking*. It corresponds to a coherent execution. The set of non-blocking traces on A is $\mathcal{T}_A$. $\mathcal{T}_\emptyset$ is the singleton $\{1_\mathcal{T}\}$, where the trace $1_\mathcal{T}$ maps all instants on $1_\mathcal{V}$. The trace which maps all instants on ${}^\star V_A$ is called the *silent* trace ${}^\star V_A^\omega$. A trace distinct from the silent one is said *significant*. The set of valuations associated to a trace is defined as follows:

Definition 2.4 (Set of valuations associated to T) *Given $T \in \mathcal{T}_A$, the set of valuations $[\![T]\!]_{\mathcal{V}_A} \subseteq \mathcal{V}_A$ associated to T is defined by:*

$$[\![T]\!]_{\mathcal{V}_A} \triangleq \{V \in \mathcal{V}_A \mid \exists t \in \mathbb{N}, V = T(t)\}$$

The *restriction* of a trace $T \in \mathcal{T}_A^\#$ to $A_1 \subseteq A$ is denoted by $T_{|A_1}$.

Example 2.1 Fig 1(a) shows a trace $T \in \mathcal{T}_{\{a,b\}}$ whose restriction $T_{|\{a\}} \in \mathcal{T}_{\{a\}}$ is given on Fig. 1(b).

a	1	$\star$	4	$\star$	...		a	1	$\star$	4	$\star$	...		a	1		4		...
b	$\star$	2	3	4	...		b	$\star$	2	3	4			b	$\star$	2	3	4	

(a) $T \in \mathcal{T}_{\{a,b\}}$ (b) $T_{|\{a\}} \in \mathcal{T}_{\{a\}}$ (c) $\prod_{\{a\}}(T) \in \mathcal{F}_a$

Fig. 1: Restriction

The synchronous product between valuations is extended to traces, by the composition of valuations at each instant.

Definition 2.5 (Composition of traces) *The synchronous product of traces $\odot$ is defined by:*

$$\begin{array}{rccl} \odot: & \mathcal{T}_{A_1}^\# \times \mathcal{T}_{A_2}^\# & \to & \mathcal{T}_{A_1 \cup A_2}^\# \\ & (T_1, T_2) & \mapsto & T \text{ such that } \forall i \in \mathbb{N}, T(i) = T_1(i) \cdot T_2(i) \end{array}$$

For a given set of variables A, $(\mathcal{T}_A^\#, \odot, 1_\mathcal{T})$ is a commutative and idempotent monoid. Informally, two traces $T_1 \in \mathcal{T}_{A_1}, T_2 \in \mathcal{T}_{A_2}$ are *synchronizable* if they agree on their common variables: $T_1 \odot T_2$ is non-blocking, that is $T_{1|A_1 \cap A_2} = T_{2|A_1 \cap A_2}$.

Example 2.2 Traces given on Fig. 4(a) and 4(b) page 11 are not synchronizable: they disagree on the values of b (e.g. at first instant b cannot be both absent (t_1^a) and present with value *false* (t_1^b)). On the contrary traces T_a' and T_o' given on Fig. 6(a) and Fig. 6(b) page 13 are synchronizable: their composition is the trace given on Fig. 5(a) page 11. ◇

Only non-blocking traces in $\mathcal{T}_A$ are considered in the following.

2.2.3 Flows

Intuitively the passive observation of a system only produces significant valuations, since by definition nothing happens during the silent valuation. Nevertheless an observer is authorized to look at the system as frequently as wished for, then at instants when the system does not necessarily act significantly. These more frequent observations insert in traces a finite but not bounded number of silent valuations between significant ones: the date of valuations that occur after the inserted silent ones is increased in consequence but their relative order is preserved. Densification by observation makes traces *denser* and leads to a partial order between traces: two densifications of the same trace are said to be *equivalent*. An equivalence class admits a minimal trace for the densification order which is called a *flow*. Flows only keep in traces significant valuations and their causality.

[4] It is also possible to design a semantics using *finite* traces

Densification, Partial Order between Traces A *densification function* is a monotonous function from $\mathbb{N}$ to $\mathbb{N}$. Let $T \in \mathcal{T}_A$ and f be a densification function. Then the *densification* of T by f, denoted by $f \uparrow T$, is such that:

$$\forall t \in \mathbb{N}, (f \uparrow T)(t) = \begin{cases} T(f^{-1}(t)) \text{ if } t \in f(\mathbb{N}) \\ {}^\star V_A \text{ else} \end{cases}$$

Example 2.3 The densification function f such that $f(1) = 2$, $f(2) = 3$, $f(3) = 4$, $f(4) = 7$, $f(5) = 8$, and $f(6) = 10$ densifies T into T_1 on Fig. 2. ◇

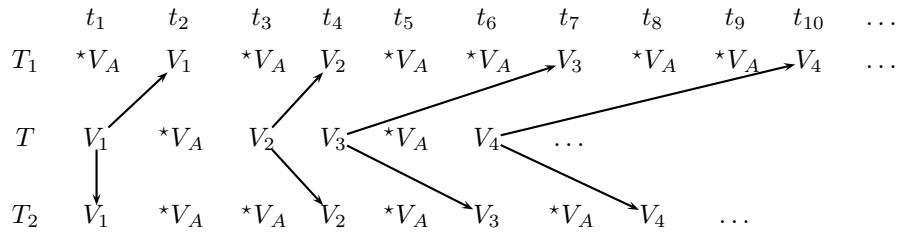

Fig. 2: Densification of traces

In accordance with what was said before, a densification function inserts a finite but not bounded number of silent valuations in traces. It therefore preserves significance of traces (which would not be the case if the number of inserted silent valuations could be infinite).

Lemma 2.1 (Densification function and composition) *Given a densification function f, and traces $T_1 \in \mathcal{T}_{A_1}$, $T_2 \in \mathcal{T}_{A_2}$:*

$$f \uparrow (T_1 \odot T_2) = (f \uparrow T_1) \odot (f \uparrow T_2)$$

Traces are partially ordered by densification according to the principle "the denser the greater". A trace T_2 is greater than a trace T_1 if it is obtained by inserting silent valuations into T_1.

Definition 2.6 (Partial order between traces) *For $T_1, T_2 \in \mathcal{T}_A$, $T_1 \leq_\star T_2$ if there exists a densification function f such that $T_2 = f \uparrow T_1$.*

Two traces are equivalent if they are the densification of a common trace.

Definition 2.7 (Equivalence between traces) *For $T_1, T_2 \in \mathcal{T}_A$, $T_1 \equiv_\star T_2$ if there exists $T \in \mathcal{T}_A$ such that $T \leq_\star T_1$ and $T \leq_\star T_2$. The equivalence class of T is denoted by $\overline{T}$.*

Remark 2.1 Note that $\leq_\star$ is reflexive. Then

$$\text{for all } T, T_1 \in \mathcal{T}_A \text{ and } T' \in \overline{T},\ T_1 \leq_\star T' \text{ implies } T_1 \in \overline{T} \tag{1}$$

Indeed $T_1 \leq_\star T_1$ and $T_1 \leq_\star T'$, therefore $T_1 \equiv_\star T'$ and $T_1 \in \overline{T}$. ◇

Example 2.4 On Fig. 2, $T \leq_\star T_1$, $T \leq_\star T_2$, therefore $T_1, T_2 \in \overline{T}$ and $T \equiv_\star T_1 \equiv_\star T_2$. ◇

Flows The "most significant" traces are now considered: they contain no silent valuations between two significant ones.

Property 1 For all $T \in \mathcal{T}_A$, $\overline{T}$ has a minimal element for $\leq_\star$, denoted by $\bigwedge_\star \overline{T}$.

Proof Assume that there exist two non-comparable traces T_1 and T_2 such that for all $T' \in \overline{T}$, $T_1 \leq_\star T'$ and $T_2 \leq_\star T'$. By Eq. (1), $T_1, T_2 \in \overline{T}$. Then by definition of $\equiv_\star$ there exists $T'' \in \mathcal{T}_A$ such that $T'' \leq_\star T_1$ and $T'' \leq_\star T_2$. Since by Eq. (1) $T'' \in \overline{T}$, we obtain a contradiction. □

A flow is a particular trace, namely the minimal element of an equivalence class of traces.

Definition 2.8 (Flow) *The set of flows on A is denoted by $\mathcal{F}_A$, with typical elements F, F_1, F', etc. When $A = \{a\}$, we simply write $\mathcal{F}_a$. A trace $T \in \mathcal{T}_A$ is a flow if it is the minimal element of $\overline{T}$.*

Property 2 Let F be a flow in $\mathcal{F}_A$. Then:

- if there exists $n \in \mathbb{N}$ such that $F(n) = {}^\star V_A$ then for all $t \geq n$, $F(t) = {}^\star V_A$;
- if for some $n \in \mathbb{N}$, $F(n) \neq {}^\star V_A$, then for all $t \leq n$, $F(t) \neq {}^\star V_A$;

A flow is then a particular trace T which is either silent ($T(0) = {}^\star V_A$), or entirely composed of significant valuations ($\forall t \in \mathbb{N}, T(t) \neq {}^\star V_A$), or else as shown on Fig. 3: there exists $n \in \mathbb{N}$ such that first instants ($0 \leq t \leq n$) correspond to significant valuations only, while "last" instants ($t > n$) correspond to an infinite sequence of silent valuations (the flow is said to *terminate*). Intuitively, a flow is obtained from a trace by erasing all silent valuations that precede a significant one.

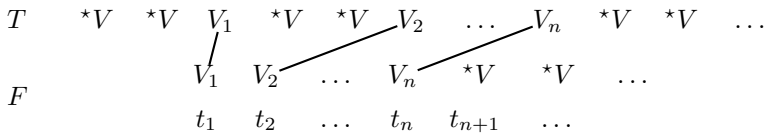

Fig. 3: A typical flow F obtained from a trace T

The restriction of a flow is more complicated than the restriction of a trace: the minimal element must be computed after the projection.

Definition 2.9 (Restriction of a flow) *The restriction of $F \in \mathcal{F}_A$ to $A_1 \subseteq A$ is defined by*

$$\prod_{A_1}(F) = \bigwedge_\star \overline{F_{|A_1}}$$

Example 2.5 Fig 1(a) on page 7 shows a flow $T \in \mathcal{F}_{\{a,b\}}$ whose restriction $\prod_{\{a\}}(T) \in \mathcal{F}_a$ is given on Fig. 1(c). ◇

Informally two flows $F_1 \in \mathcal{F}_{A_1}$ and $F_2 \in \mathcal{F}_{A_2}$ are *synchronizable* if $\prod_{A_1 \cap A_2}(F_1) = \prod_{A_1 \cap A_2}(F_2)$. Like for traces, this notion involves restriction: the two flows must agree on the order and on the values of their common variables, but unlike traces they do not need to agree on their exact instant of presence. Therefore synchronous product is not the good tool for flows.

Example 2.6 Like for Ex 2.2 the synchronous product of flows F_1 and F_2 shown on Fig. 4(a) and 4(b) on page 11 is blocking. Nevertheless the two flows disagree only on the instants of presence of b: they agree on its sequence of values *false*, *true*, *false*. Therefore there exist $T_1 \in \overline{F_1}$, $T_2 \in \overline{F_2}$ s.t. T_1 and T_2 are synchronizable. Indeed $\prod_{A_1 \cap A_2}(F_1) = \prod_{A_1 \cap A_2}(F_2)$ and F_1 and F_2 are synchronizable. ◇

2.2.4 Clocks

As defined informally in Sect. 2.1, clocks are subsets of the time referential. A typical example is the clock of a variable $x \in S$, denoted by $\widehat{x}$, which is the set of its instants of presence; but the set of instants at which x and y are present and their sum is positive is also a clock (which can be indirectly represented by the clock of a variable). A set of instants alone has no meaning: the clock of x should be defined with respect to a given observation of the system which provides a local time referential. According to the chosen semantics, such an observation is a trace or a flow. We use traces in this section: the counterpart for flows is immediate since flows are particular traces. A clock is defined on the basis of a given trace by means of its *characteristic function*. Generally speaking, the characteristic function of a subset $X' \subseteq X$ defines X' by mapping any element $x \in X$ to *true* if $x \in X'$, else to *false*. Similarly the clock of the variable x w.r.t. a trace T, denoted by $\widehat{x}_T$, is defined by examining valuations $T(t)$ for any instant $t \in \mathbb{N}$. If the valuation denotes an instant of the clock of x (i.e. $T(t)(x) \neq \star$) then $t \in \widehat{x}_T$ and $T(t)$ is said to *switch on* $\widehat{x}_T$. If $t \notin \widehat{x}_T$, $T(t)$ is said to *switch off* $\widehat{x}_T$.

Definition 2.10 (Clock of a variable w.r.t. a trace) *For $A \subseteq S$, $T \in \mathcal{F}_A$ and $x \in A$, the clock of x in T is written $\widehat{x}_T$ and is defined as the least subset of $\mathbb{N}$ s.t.*

$$\forall t \in \mathbb{N},\ t \in \widehat{x}_T \text{ iff } T(t)(x) \neq \star.$$

A natural pre-order $\subseteq_T$ between clocks (w.r.t. to the trace T) is derived accordingly: $\widehat{x} \subseteq_T \widehat{y}$ iff $\widehat{x}_T \subseteq \widehat{y}_T$. The expansion of this definition in terms of characteristic function shows that in accordance this intuition, $\widehat{x} \subseteq_T \widehat{y}$ iff for any valuation $T(t)$, $T(t)$ switches on $\widehat{x}$ implies that $T(t)$ also switches on $\widehat{y}$:

$$\begin{aligned} \widehat{x} \subseteq_T \widehat{y} \quad & \text{iff } \forall t \in \mathbb{N},\ t \in \widehat{x}_T \text{ implies } t \in \widehat{y}_T \\ & \text{iff } \forall t \in \mathbb{N},\ T(t)(x) \neq \star \text{ implies } T(t)(y) \neq \star \end{aligned}$$

Thus $\widehat{x} \subseteq_T \widehat{y}$ expresses a property of the trace which is true for all instant t (thus it is a safety property) and describes only what happens inside the instant, with no reference to instants which precede or follow it: it is an *instantaneous property* of the trace, as introduced in Sect. 1. Two variables x and y are *synchronous* (in T) if $\widehat{x} =_T \widehat{y}$, or $\widehat{x}_T = \widehat{y}_T$.

Clock Algebra Associated to a Trace Clocks can be described by the language $\langle U, \cap, \cup, \setminus \rangle$, where U is a symbol of constant and $\cup, \cap$ and $\setminus$ are symbols of relation. Assume given a finite set of variables $\mathcal{K}$. The language is interpreted into *clock algebras* when U and variables in $\mathcal{K}$ are interpreted as sets of instants: U denotes the time referential and elements of $\mathcal{K}$ denote subsets of U. Symbols $\cap$, $\cup$ and $\setminus$ are interpreted as classical set operators. We use the notation $\mathbb{O}$ to denote $U \setminus U$, that is the *empty* or *null* clock.

The clock algebra associated to a trace $T \in \mathcal{T}_A$ is obtained as follows. U is interpreted as the local time referential denoted by T, that is the set of instants $t \in \mathbb{N}$ at which $T(t)$ is significant:

$$U = \{t \in \mathbb{N} \mid T(t) \neq {}^{\star}V_A\}$$

Then if the variable $\widehat{x} \in \mathcal{K}$ denotes the clock of the variable $x \in A$, $\widehat{x}$ is interpreted as $\widehat{x}_T$, and $\mathbb{O}$ as the empty set.

2.3 Processes

Section 2.3.1 gives notations and intuitive semantics for processes. Their semantics in terms of traces (resp. flows) is given in Sect. 2.3.2 (resp. in Sect. 2.3.3). A useful theorem is given in Sect. 2.3.4, finally clocks are dealt with in Sect. 2.3.5.

2.3.1 Informal Presentation

The semantics of a data-flow specification on a set of variables A is given by a *process*. The set of processes on A is denoted by $\mathcal{P}_A$, whose typical elements are ϖ, ϖ_1, etc. A process describes the possible behaviors of the specification: it is a set of models each of ones represents a possible execution. It can therefore be seen as a set of traces or a set of flows. Both semantics are equivalent: the flow one is well adapted to the description of classes of processes (see Sect. 2.4) and data-flow operators (see Sect. 3.1) while a weaker trace semantics is commonly used to describe operational semantics (see Sect. 2.5).

Example 2.7 Processes described in this example will be used in the whole section 2.3. Let us consider a boolean variable b and four integer variables a_1, a_2, o_1, o_2. The process $\varpi_{ba} \in \mathcal{P}_{\{b,a_1,a_2\}}$ acts as follows:

- when present b carries alternatively values *true* and *false*;
- if b is present then a_2 is absent; if b is absent then a_2 is absent or present, in this case it carries a free value in $\mathbb{N}$;
- if b is present with value *true* then a_1 is present and carries a free value in $\mathbb{N}$; if b is absent or present with value *false* then a_1 is absent.

The process $\varpi_{bo} \in \mathcal{P}_{\{b,o_1,o_2\}}$ acts as follows:

- when present b carries alternatively values *true* and *false*;
- if b is present then o_2 is absent; if b is absent then o_2 is absent or present, in this case it carries a free value in $\mathbb{Z}$;
- if b is present with value *false* then o_1 is present and carries a free value in $\mathbb{Z}$; if b is absent or present with value *true* then o_1 is absent.

A behavior of ϖ_{ba} (resp. ϖ_{bo}) is given on Fig. 4(a) (resp. Fig. 4(b)) where T (resp. F) is written for *true* (resp. *false*). ◇

	t_1^a	t_2^a	t_3^a	t_4^a	t_5^a	t_6^a	...
b	$\star$	$\star$	F	T	$\star$	F	...
a_1	$\star$	$\star$	$\star$	2	$\star$	$\star$	...
a_2	12	13	$\star$	$\star$	3	$\star$	...

(a) A behavior of ϖ_{ba}

	t_1^b	t_2^b	t_3^b	t_4^b	t_5^b	...
b	F	T	$\star$	$\star$	F	...
o_1	1	$\star$	$\star$	$\star$	6	...
o_2	$\star$	$\star$	4	5	$\star$	...

(b) A behavior of ϖ_{bo}

Fig. 4: Behavior for processes

Because processes are sets, *inclusion* and *equivalence* of processes are just set inclusion and set equivalence. Operations of restriction and parallel composition already presented for traces and flows are extended to processes. Their precise semantics will be given for sets of traces (resp. flows) in Sect. 2.3.2 (resp. Sect. 2.3.3): let us describe informally here the restriction, then the parallel composition.

Restriction The *restriction* of $\varpi \in \mathcal{P}_A$ to $A_1 \subseteq A$ is a process on A_1 denoted by $\prod_{A_1}(\varpi)$. Like for traces and flows, the restriction to A_1 lets visible only variables of A_1. Variables of $A \setminus A_1$ are hidden and correspond intuitively to variables local to ϖ.

Example 2.8 If one considers only the variable b in ϖ_{bo} and ϖ_{ba}, then processes behave similarly: both alternate values *true* and *false*. Then $\prod_{\{b\}}(\varpi_{bo}) = \prod_{\{b\}}(\varpi_{ba})$. ◇

Parallel Composition Classically a synchronous specification makes act in parallel two or more smallest subspecifications (or modules). It corresponds semantically to the *parallel composition* of processes. The parallel composition of two processes $\varpi_1 \in \mathcal{P}_{A_1}$ and $\varpi_2 \in \mathcal{P}_{A_2}$ is denoted by $\varpi_1 \parallel \varpi_2 \in \mathcal{P}_{A_1 \cup A_2}$, where $\parallel$ is the *operator* of parallel composition. $\varpi_1 \parallel \varpi_2$ contains behaviors of both ϖ_1 and ϖ_2 provided they agree on their common variables in $A_1 \cap A_2$: ϖ_1 and ϖ_2 communicate and synchronize each others by these variables. A behavior of $\varpi_1 \parallel \varpi_2$ restricted to A_1 (resp. A_2) is therefore a behavior of ϖ_1 (resp. ϖ_2).

Example 2.9 Let us denote by ϖ_{boa} the process $\varpi_{bo} \parallel \varpi_{ba} \in \mathcal{P}_{\{b,a_1,a_2,o_1,o_2\}}$. ϖ_{boa} acts as follows:

- when present b carries alternatively values *true* and *false*;
- if b is present then a_2 *and* o_2 are absent; if b is absent then a_2 is absent or present, in this case it carries a free value in $\mathbb{Z}$, *and* so does o_2 ;
- if b is present with value *false* then o_1 is present and carries a free value in $\mathbb{Z}$ *and* a_1 is absent; if b is present with value *true* then o_1 is absent *and* a_1 is present and carries a free value in $\mathbb{Z}$. If b is absent then a_1 *and* o_1 are absent.

Two possible behaviors of ϖ_{boa} are given in Fig. 5: they correspond to the composition of behaviors for ϖ_{ba} and ϖ_{bo} given on Fig. 4(a) and Fig. 4(b). They differ from the instant at which a_2 carries the value 3: t_5 or t_6.
Let us now define the process $\varpi_{boF} \in \mathcal{P}_{\{b,o\}}$ (resp. $\varpi_{boT} \in \mathcal{P}_{\{b,o\}}$) which has the same behavior as ϖ_{bo} except

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
a_2	12	13	$\star$	$\star$	3	$\star$	$\star$	...
a_1	$\star$	$\star$	$\star$	2	$\star$	$\star$	$\star$	...
b	$\star$	$\star$	F	T	$\star$	$\star$	F	...
o_1	$\star$	$\star$	1	$\star$	$\star$	$\star$	6	...
o_2	$\star$	$\star$	$\star$	$\star$	4	5	$\star$	...

(a)

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
a_2	12	13	$\star$	$\star$	$\star$	3	$\star$	...
a_1	$\star$	$\star$	$\star$	2	$\star$	$\star$	$\star$	...
b	$\star$	$\star$	F	T	$\star$	$\star$	F	...
o_1	$\star$	$\star$	1	$\star$	$\star$	$\star$	6	...
o_2	$\star$	$\star$	$\star$	$\star$	4	5	$\star$	...

(b)

Fig. 5: Two possible behaviors for ϖ_{boa}

that the first value carried by b is necessarily *false* (resp. *true*). Processes $\varpi_{baF} \in \mathcal{P}_{\{b,a\}}$ and $\varpi_{baT} \in \mathcal{P}_{\{b,a\}}$ are defined similarly. Then:

- The process $\varpi_{boaF} \in \mathcal{P}_{\{b,a_1,a_2,o_1,o_2\}}$ (resp. $\varpi_{boaT} \in \mathcal{P}_{\{b,a_1,a_2,o_1,o_2\}}$) defined by $\varpi_{boF} \parallel \varpi_{baF}$ (resp. $\varpi_{boT} \parallel \varpi_{baT}$) acts like ϖ_{boa} except that the first value carried by b is necessarily *false* (resp. *true*);
- On the contrary, the process ϖ_{boTaF} (resp. ϖ_{boFaT}) defined by $\varpi_{boT} \parallel \varpi_{baF}$ (resp. $\varpi_{boF} \parallel \varpi_{baT}$) cannot behave significantly if b is present: neither ϖ_{boT} and ϖ_{baF} nor ϖ_{boF} and ϖ_{baT} can agree on its first value. b is then constrained to be always absent in ϖ_{boTaF} and ϖ_{boFaT}: only a_2 and o_2 can be present;
- Let us now consider only a part of specifications and forget a_2 and o_2: let us assume that ϖ_{ba} (resp. ϖ_{bo}) is defined on $\{b, a_1\}$ (resp. $\{b, o_1\}$). In this case ϖ_{boTaF} and ϖ_{boFaT} cannot act significantly at all: their set of *significant* behaviors is empty. Nevertheless processes cannot be empty: a process that cannot act significantly contains exactly the silent behavior ${}^\star V_A^\omega$.

◇

Let us now focus on the properties of parallel composition. Two processes $\varpi_1 \in \mathcal{P}_{A_1}$ and $\varpi_2 \in \mathcal{P}_{A_2}$ are informally *synchronizable at a given instant* if they can act with agreement on their common variables $A_1 \cap A_2$. If ϖ_1 cannot synchronize with ϖ_2 at a given instant then it *stutters* in the meanwhile: it "carries out" a silent valuation (better said ϖ_1 is observed at an instant where it does not act significantly) while ϖ_2 acts significantly. Processes are always authorized to stutter while waiting for synchronization: a process is said to be *closed under stuttering*. On the other hand, synchronization only involves a finite but not bounded stuttering: a process cannot stutter infinitely if waiting for a possible synchronization. Two processes that can synchronize will eventually do so: parallel composition is *fair* and leads to *true parallelism*.

Example 2.10 Let us consider actions of Fig. 4. At instant t_b^1 ϖ_{bo} can synchronize with ϖ_{ba} if b is present with value *false*. But at instants t_1^a and t_2^a b is absent in ϖ_{ba}: synchronization can only occur at instant t_3^a. Since ϖ_{bo} and ϖ_{ba} cannot synchronize at instants t_1 and t_2, ϖ_{bo} *stutters* while ϖ_{ba} acts significantly: it "carries out" two silent valuations (see Fig. 5). But ϖ_{bo} is not allowed to stutter infinitely.
Assume now that at t_3^a b is present with value *true*. Then synchronization between these two behaviors is impossible: either processes both stutter infinitely, or according to notations of Sect. 2.2.1 and 2.2.2 there is a blocking #. On the one hand behaviors we consider are non-blocking; on the other hand we must semantically be able to make the difference between blocking behaviors (that begin normally then stutter infinitely) and an execution in which the system does not act as a normal behavior. Therefore the composition of two non-synchronizable executions do not belong to the parallel composition. If all behaviors are non-synchronizable, then the parallel composition reduces to the silent behavior. ◇

Two processes $\varpi_1 \in \mathcal{P}_{A_1}$ and $\varpi_2 \in \mathcal{P}_{A_2}$ are informally *synchronizable* if their parallel composition is not reduced to $\{{}^\star V_A^\omega\}$. This particular *silent* process resulting from a parallel composition denotes an impossible synchronization between processes that finds expression into an *infinite stuttering*. The silent process must not be confused with the process defined on the empty set of variables $1_\mathcal{P}$, neutral element of the commutative and idempotent monoid $(\mathcal{P}_A, \parallel, 1_\mathcal{P})$.

2.3.2 Processes as Closed Sets of Traces

As said informally in Sect. 2.3.1, a process is a set of behaviors *closed under stuttering*. When models of behaviors are traces, stuttering corresponds to densification of traces: given $A \subseteq S$ and $\varpi \in \mathcal{P}_A$, if $T \in \varpi$ then $\overline{T} \subseteq \varpi$. A process then contains a trace for *each* possible observation of the system. Formally,

Definition 2.11 (Process as set of traces) *Given $A \subseteq S$, a set of traces $\varpi \subseteq \mathcal{T}_A$ is a process if*

$$\varpi = \bigcup\{\overline{T} \mid T \in \varpi\}$$

It is easy to verify that the function $\varpi \mapsto \bigcup\{\overline{T} \mid T \in \varpi\}$ is a closure operator[5] on $\mathcal{T}_A$ One must ensure that operations on processes as sets of traces preserve their closure under stuttering. It is trivial for restriction:

[5] Formally, a closure operator $\phi : \mathcal{P}(X) \to \mathcal{P}(X)$ on a set X is *extensive*: $\forall X' \subseteq X,\ X' \subseteq \phi(X')$; *idempotent*: $\forall X' \subseteq X,\ \phi(X') = \phi(\phi(X'))$ and *monotonous*: $\forall X_1, X_2 \subseteq X,\ X_1 \subseteq X_2$ implies $\phi(X_1) \subseteq \phi(X_2)$

Definition 2.12 (Restriction of a process as set of traces) *Given $A_1 \subseteq A \subseteq S$, the restriction to A_1 of $\varpi \in \mathcal{P}_A$ is defined by*

$$\prod_{A_1}(\varpi) = \{T_{|A_1} \mid T \in \varpi\}.$$

Let us now define the synchronous parallel composition between $\varpi_1 \in \mathcal{P}_{A_1}$ and $\varpi_2 \in \mathcal{P}_{A_2}$. The intuition is very natural using the synchronous product of traces. Informally $\varpi_1 \parallel \varpi_2$ contains the synchronous product $T_1 \odot T_2$ of any synchronizable traces $T_1 \in \varpi_1$ and $T_2 \in \varpi_2$. Then we can write

$$\varpi_1 \parallel \varpi_2 = \{T \in \mathcal{T}_{A_1 \cup A_2} \mid \text{ there exist } T_1 \in \varpi_1, T_2 \in \varpi_2 \text{ s.t. } T_{1|A_1 \cap A_2} = T_{2|A_1 \cap A_2} \text{ and } T = T_1 \odot T_2\} \quad (2)$$

Said differently, any trace of $\varpi_1 \| \varpi_2$ is *a posteriori* the result of the composition of a certain trace of ϖ_1 with a certain trace of ϖ_2.

Definition 2.13 (Composition of processes as sets of traces) *Given $A_1, A_2 \subseteq S$, $\varpi_1 \in \mathcal{P}_{A_1}$ and $\varpi_2 \in \mathcal{P}_{A_2}$,*

$$\varpi_1 \| \varpi_2 = \{T \in \mathcal{T}_{A_1 \cup A_2} \mid T_{|A_1} \in \varpi_1 \text{ and } T_{|A_2} \in \varpi_2\}.$$

It is not obvious that the above definition preserves closure under stuttering. Let us consider some traces $T \in \varpi_1 \| \varpi_2$ and $T' \in \overline{T}$ and prove that $T' \in \varpi_1 \| \varpi_2$. By (2) there exist $T_1 \in \varpi_1$ and $T_2 \in \varpi_2$ s.t. $T = T_1 \odot T_2$. By Def. 2.7 and 2.6, there exist a trace $T'' \in \overline{T}$ and a densification function f (resp. f') such that $f \uparrow T'' = T$ (resp. $f' \uparrow T'' = T'$). Let us write $T'_1 = T'_{|A_1}$, $T'_2 = T'_{|A_2}$, $T''_1 = T''_{|A_1}$ and $T''_2 = T''_{|A_2}$. Then $T' = T'_1 \odot T'_2$ (resp. $T'' = T''_1 \odot T''_2$) and by Lem. 2.1 $f \uparrow T''_1 = T_1$ (resp. $f \uparrow T''_2 = T_2$) and $f' \uparrow T''_1 = T'_1$ (resp. $f' \uparrow T''_2 = T'_2$). Therefore $T_1, T'_1 \in \overline{T''_1}$ (resp. $T_2, T'_2 \in \overline{T''_2}$) and $T_1 \equiv_\star T'_1$ (resp. $T_2 \equiv_\star T'_2$). But since ϖ_1 (resp. ϖ_2) is a process at the sense of Def. 2.11, $T'_1 \in \varpi_1$ (resp. $T'_2 \in \varpi_2$). Because $T' = T'_1 \odot T'_2$, $T' \in \varpi_1 \parallel \varpi_2$ by (2).

Example 2.11 Let us consider again ϖ_{boa} given in Ex. 2.9 on page 11 and denote by $T_a \in \varpi_{ba}$ (resp. $T_o \in \varpi_{bo}$) the behaviors given on Fig. 4(a) (resp. 4(b)). T_a and T_o are not synchronizable: their synchronous product produces a blocking. But ϖ_{ba} contains traces T'_a and T''_a shown on Fig. 6(a), that are densifications of T_a; and ϖ_{bo} contains the trace T'_o shown on Fig. 6(b), that is a densification of T_o. $T'_a \odot T'_o$ is exactly the trace shown on Fig. 5(a) while $T''_a \odot T'_o$ is exactly the trace shown on Fig. 5(b).

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
a_2	12	13	$\star$	$\star$	3	$\star$	$\star$	...
a_1	$\star$	$\star$	$\star$	2	$\star$	$\star$	$\star$	...
b	$\star$	$\star$	F	T	$\star$	$\star$	F	...

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
a_2	12	13	$\star$	$\star$	$\star$	3	$\star$	...
a_1	$\star$	$\star$	$\star$	2	$\star$	$\star$	$\star$	...
b	$\star$	$\star$	F	T	$\star$	$\star$	F	...

(a) T'_a at the left, T''_a at the right

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
b	$\star$	$\star$	F	T	$\star$	$\star$	F	...
o_1	$\star$	$\star$	1	$\star$	$\star$	$\star$	6	...
o_2	$\star$	$\star$	$\star$	$\star$	4	5	$\star$	...

(b) T'_o

Fig. 6: Densified traces for Ex. 2.11

2.3.3 Processes as Sets of Flows

Since a flow F is the minimal element of the set $\overline{F}$, it is implicitly closed under stuttering, without enumeration of all possible observations.

Definition 2.14 (Process as a set of flows) *Given $A \subseteq S$, a process on A is any subset of $\mathcal{F}_A$.*

Firstly the restriction is defined similarly as it was for traces:

Definition 2.15 (Restriction of a process as a set of flows) *Given $A_1 \subseteq A \subseteq S$, the restriction of $\varpi \in \mathcal{P}_A$ is defined by*

$$\prod_{A_1}(\varpi) = \{\prod_{A_1}(F) \mid F \in \varpi\}.$$

The definition of parallel composition follows the parallel composition of sets of traces:

Definition 2.16 (Composition of processes as sets of flows) *Given $A_1, A_2 \subseteq S$, $\varpi_1 \in \mathcal{P}_{A_1}$ and $\varpi_2 \in \mathcal{P}_{A_2}$,*

$$\varpi_1 \| \varpi_2 = \{F \in \mathcal{F}_{A_1 \cup A_2} \mid \prod_{A_1}(F) \in \varpi_1 \text{ and } \prod_{A_2}(F) \in \varpi_2\}$$

The intuition is nevertheless quite different from the case of sets of traces. Of course, it is always possible to densify flows then to reason in terms of traces. Another intuition is highlighted in Ex. 2.12. Informally to obtain all flows in $\varpi_1 \parallel \varpi_2$ one should try all pairs $(F_1, F_2) \in \varpi_1 \times \varpi_2$. If F_1 and F_2 are synchronizable, let us denote by F their restriction to $A_1 \cap A_2$. Flows $F_{1,2} \in \varpi_1 \| \varpi_2$ induced by the composition of F_1 and F_2 are built from F as follows:
1. Let us consider pairs of valuations (V_1, V_2) such that V_1 (resp. V_2) appears in F_1 (resp. F_2) and the pair participates to F (meaning that $V_{1|A_1 \cap A_2} = V_{2|A_1 \cap A_2}$ and $V_{1|A_1 \cap A_2} \neq {}^\star V_{A_1 \cap A_2}$). Then their synchronous product $V_1 \cdot V_2$ appears in $F_{1,2}$.
2. Let us now consider the others valuations V_1' (resp. V_2') that appear in F_1 (resp. F_2), i.e. s.t. $V'_{1|A_1 \cap A_2} = V'_{2|A_1 \cap A_2} = {}^\star V_{A_1 \cap A_2}$). Inside F_1 (resp. F_2) such valuations form sub-sequences W_1^i (resp. W_2^j) that appear between two valuations mentioned at step 1. Inside a pair (W_1^i, W_2^i) valuations that appear in W_1^i and valuations that appear in W_2^i are not constrained between themselves since variables they share are absent. Sub-sequences $W_{1,2}^i$ that appear in $F_{1,2}$ between two valuations mentioned at step 1 result from the composition $W_1^i \otimes W_2^i$ defined as follows. If *nil* denotes the empty sequence, then $nil \otimes W_2$ (resp. $W_1 \otimes nil$) is obtained by replacing in W_2 (resp. W_1) any valuation V_2 (resp. V_1) by ${}^\star V_{A_1 \setminus A_2} \cdot V_2$ (resp. $V_1 \cdot {}^\star V_{A_2 \setminus A_1}$). Then if *hd* (resp. *tl*) denotes the head (resp. the tail) of a sequence:

$$(1)\ \begin{array}{l} hd(W_1 \otimes W_2) = hd(W_1) \cdot hd(W_2) \\ tl(W_1 \otimes W_2) = tl(W_1) \otimes tl(W_2) \end{array} \text{ or } (2)\ \begin{array}{l} hd(W_1 \otimes W_2) = hd(W_1) \cdot {}^\star V_{A_2 \setminus A_1} \\ tl(W_1 \otimes W_2) = tl(W_1) \otimes W_2 \end{array} \text{ or}$$

$$(3)\ \begin{array}{l} hd(W_1 \otimes W_2) = {}^\star V_{A_1 \setminus A_2} \cdot hd(W_2) \\ tl(W_1 \otimes W_2) = W_1 \otimes tl(W_2) \end{array}$$

Example 2.12 Let us consider again ϖ_{boa} given in Ex. 2.9 and denote by $F_a \in \varpi_{ba}$ (resp. $F_o \in \varpi_{bo}$) the behaviors given on Fig. 4(a) (resp. 4(b)). F_a and F_o are synchronizable and let us denote by $F \in \mathcal{F}_b$ their restriction to $\{b\}$. A flow $F_{boa} \in \varpi_{boa}$ is then built from F as follows and as illustrated on Fig. 7.
1. pairs of valuations whose synchronous product appears in F_{boa} are $(F_a(t_3^a), F_b(t_1^b))$, $(F_a(t_4^a), F_b(t_2^b))$ and $(F_a(t_6^a), F_b(t_5^b))$ are indicated by dashed lines.
2. the pair of subsequences $(F_a(t_1^a)F_a(t_2^a), nil)$ produces the subsequence $F_{boa}(t_1)F_{boa}(t_2)$. The pair of subsequences $(F_a(t_a^5), F_b(t_3^b)F_b(t_4^b))$ produces possibly the two subsequences $F_{boa}(t_5)F_{boa}(t_6)$ indicated by two boxes on Fig. 7: the first one is obtained by application of case (1), the second by application of cases (3) then (1). Three other possibilities are not represented: if V_3 denotes $F_a(t_3^a) \cdot {}^\star V_{\{o_1,o_2\}}$, V_4 denotes $F_b(t_3^b) \cdot {}^\star V_{\{a_1,a_2\}}$ and V_5 denotes $F_b(t_4^b) \cdot {}^\star V_{\{a_1,a_2\}}$ then possible sequences are $V_3V_4V_5$, $V_4V_3V_5$ and $V_4V_5V_3$. ◇

2.3.4 A Useful Theorem

Since the parallel composition corresponds to the intersection of sets of models, it is possible to express the inclusion of two processes by the equivalence between one of them and their composition. The following theorem will be widely used in Sect. 5.1.

Theorem 2.1 *For $A' \subseteq A \subseteq S$, $\varpi \in \mathcal{P}_A$ and $\varpi' \in \mathcal{P}_{A'}$,*

$$\varpi = \varpi \| \varpi' \text{ iff } \prod_{A'}(\varpi) \subseteq \varpi'$$

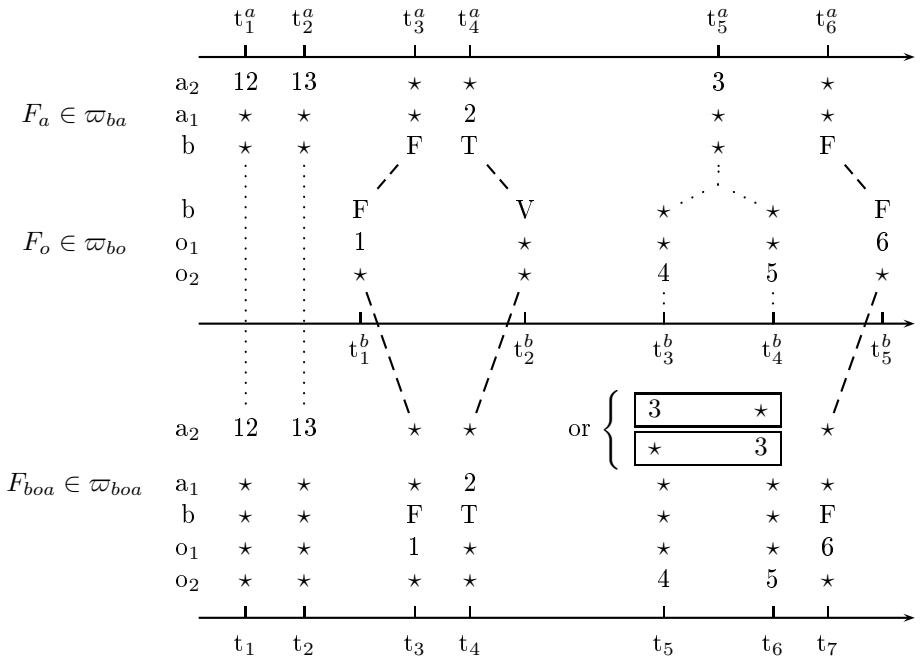

Fig. 7: Parallel Composition

Proof We consider that processes are set of flows.
$\Rightarrow$) Assume that $\varpi = \varpi \| \varpi'$ and let us consider $F' \in \prod_{A'}(\varpi)$. Necessarily by Def 2.15 there exists $F \in \varpi$ such that $F' = \prod_{A'}(F)$. Moreover by hypothesis and by definition of the parallel composition (Def. 2.16) $\prod_A(F) \in \varpi$ and $\prod_{A'}(F) = F' \in \varpi'$.
$\Leftarrow$) Assume $\prod_{A'}(\varpi) \subseteq \varpi'$. If $F \in \varpi$ then $\prod_{A'}(F) \in \varpi'$, and since $\prod_A(F) = F$, we have $F \in \varpi \| \varpi'$. Conversely, if $F \in \varpi \| \varpi'$, then $\prod_A(F) = F \in \varpi$.

□

2.3.5 Clocks

The pre-order between clocks presented in Sect. 2.2.4 is naturally extended to processes, considered as sets of traces:

Definition 2.17 (Pre-order between clocks in a process) *For $A \subseteq S$, $\varpi \in \mathcal{P}_A$ and $x, y \in A$, the inclusion of the clock of x into the clock of y is denoted by $x \subseteq_\varpi y$ and defined by:*

$$x \subseteq_\varpi y \text{ iff for all } T \in \varpi, \widehat{x} \subseteq_T \widehat{y}$$

Rewriting this definition like in Sect. 2.2.4 we obtain:

$$x \subseteq_\varpi y \text{ iff } \forall T \in \varpi, \forall t \in \mathbb{N},\ T(t)(x) \neq \star \text{ implies } T(t)(y) \neq \star$$

Therefore an inclusion between clocks is an *instantaneous property* of a process, true in any trace (or flow) of the process. x and y are said to be synchronous in ϖ if $\forall T \in \varpi,\ \widehat{x} =_T \widehat{y}$.

2.4 Classes of Processes

A process $\varpi \in \mathcal{P}_A$ is said to be *monochronous* if all variables in A are synchronous in ϖ. Otherwise it is said to be *polychronous*.

In the following of this section S is split into I and O, where I is the set of input variables assumed to be totally ordered and O is the set of output variables (local variables are neglected). **The semantics for processes in $\mathcal{P}_{I \cup O}$ is given as a set of flows.**

2.4.1 Deterministic Processes on I

Informally a process $\varpi \in \mathcal{P}_{I \cup O}$ is deterministic if any flow $F \in \varpi$ is entirely determined by its restriction to input variables I.

Definition 2.18 (Deterministic Process) *A process $\varpi \in \mathcal{P}_{I \cup O}$ is* deterministic *on I if the function*

$$\begin{array}{lcl} \mathcal{F}_{I \cup O} & \rightarrow & \mathcal{F}_I \\ F & \mapsto & \prod_I(F) \end{array}$$

is bijective on ϖ.

As an example, Ex. 2.13 gives two versions of a counter, a deterministic and a non-deterministic one.

Example 2.13 Assume that $A = \{a, ma, N\}$ is composed of integer variables. Assume given a process $\varpi \in \mathcal{P}_A$ that describes a counter as follows: the variable a decrements and reinitializes itself after it has reached 0, with the positive input variable N. So N is needed only after a has reached value 0. Variables a and ma are synchronous. The variable ma carries the previous value of a, initially 0. So N is synchronized with instants where $ma = 0$. ϖ is deterministic on $\{N\}$. A flow of ϖ is given in Fig.8(a). Its restriction to $\{N\}$ (i.e. the sequence of values beginning with 2,1,3) entirely determines the flow on A. Let us now define the process

	t_1	t_2	t_3	t_4	t_5	t_6	...
N	2	$\star$	$\star$	1	$\star$	3	...
a	2	1	0	1	0	3	...
ma	0	2	1	0	1	0	...

(a) A flow of ϖ

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	...
N	2	$\star$	$\star$	$\star$	$\star$	1	$\star$	3	...
a	2	1	0	-1	-2	1	0	3	...
ma	0	2	1	0	-1	-2	1	0	...

(b) A flow of ϖ'

Fig. 8: A "counter"

$\varpi' \in \mathcal{P}_A$ which describes a counter in which a and ma have the same behavior as in ϖ, but the presence of N is not synchronized with a particular value of ma. Re-initialization of the counter occurs in a non-deterministic way therefore ϖ' is not deterministic. As an example: for the same sequence of inputs 2,1,3, a possible flow in ϖ' is the one of Fig.8(a), but also the one of Fig 8(b). $\diamond$

2.4.2 Endochronous Processes on I

Informally, a process $\varpi \in \mathcal{P}_{I \cup O}$ is endochronous on I if any flow $F \in \varpi$ is entirely determined by the sequences of *significant values* carried by inputs variables, independently of their relative presence and absence. Therefore F is uniquely determined by the tuple of its restrictions to single variables in I. In absence of synchronization information, an endochronous process is able to reorganized this *tuple of sequences* into *sequences of valuations* (valuation meaning tuple of values), that is in a single flow. An operational characterization of *endochrony* can be found e.g. in [57].

Definition 2.19 (Endochronous Process) *A process $\varpi \in \mathcal{P}_{I \cup O}$ is* endochronous *on $I = \{i_1, i_2, \ldots, i_n\}$ if the function*

$$\begin{array}{lcl} \mathcal{F}_{I \cup O} & \rightarrow & \mathcal{F}_{i_1} \times \mathcal{F}_{i_2} \times \cdots \times \mathcal{F}_{i_n} \\ F & \mapsto & (\prod_{\{i_1\}}(F), \prod_{\{i_2\}}(F), \ldots, \prod_{\{i_n\}}(F)) \end{array}$$

is bijective on ϖ.

As an illustration, an endochronous process knows when it has to read its inputs, therefore is autonomous when run in an asynchronous environment. In comparison, non-endochronous processes must be connected to an operating system which imposes the status of inputs.

Remark 2.2 An endochronous process is deterministic. Intuitively, if it is possible to infer flows of a process from the only values of input variables, it is also possible to infer them with the additional information of their relative presence and absence. The converse does not hold since information of synchronization has been lost. ◇

Example 2.14 The process ϖ which merges two inputs variables i_1 and i_2 on an output variable o with priority given to i_1 is not endochronous on $\{i_1, i_2\}$. Figure 9(a) shows how the pair of input sequences $(1^\omega, 3^\omega)$ carried by (i_1, i_2) can be reorganized in a flow on $\{i_1, i_2, o\}$, resulting to the sequence $1, 3, 3, 3, 3 \ldots$ for o. Note that any other combination of values for i_1 and i_2 is possible. For example the flow on $\{i_1, i_2, o\}$ given on Fig. 9(b) could also be obtained, resulting to the sequence $1, 3, 1, 3, 1 \ldots$ for o. Let us now consider the process ϖ' in which two synchronous input boolean conditions C_1 and C_2 are added. The truth values of C_1 resp. C_2 indicate the presence of i_1 resp. i_2 and let no choice for reorganizing the tuple of four input sequences into a flow: ϖ' is endochronous. Note that the flow shown on Fig 9(b) is the only possible given the input sequences for C_1 and C_2: their values contain all the synchronization information.

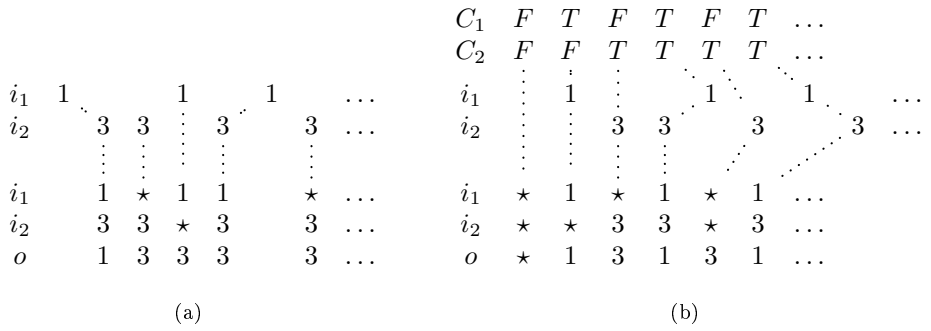

Fig. 9: A merge

◇

2.4.3 Static Processes

A process $\varpi \in \mathcal{P}_A$ is static if it is *closed under permutations of valuations*: flows are then seen as sets more than sequences of valuations. A process being a set of flows, a static process in then characterized by a set of valuations. As a consequence its behavior is *independent* of the *evolution of time*, the relation that characterizes its actions is *invariant* over the time. Let us first define the set of valuations associated to ϖ, denoted by $[\![\varpi]\!]_{\mathcal{V}_A}$, which contains all valuations that occur at any instant in any flow of ϖ.

Definition 2.20 (Set of valuations associated to ϖ) *For $\varpi \in \mathcal{P}_A$,*

$$[\![\varpi]\!]_{\mathcal{V}_A} \triangleq \bigcup_{F \in \varpi} [\![F]\!]_{\mathcal{V}_A}$$

The static closure of a process ϖ adds to ϖ any flow composed of valuations $V_i \in [\![\varpi]\!]_{\mathcal{V}_A}$.

Definition 2.21 (Static closure) *The static closure of $\varpi \in \mathcal{P}_A$ denoted by $\varpi_{\mathcal{S}} \in \mathcal{P}_A$ is defined by*

$$\varpi_{\mathcal{S}} = \{F \in \mathcal{F}_A \mid F \in [\![\varpi]\!]^{\omega}_{\mathcal{V}_A}\}$$

A process ϖ is *static* if it contains its static closure.

Definition 2.22 (Static process) *A process $\varpi \in \mathcal{P}_A$ is static if $\varpi = \varpi_{\mathcal{S}}$. The set of static processes on A is denoted by $\mathcal{P}^{\mathcal{S}}_A$.*

It is easy to verify that the function $\varpi \mapsto \varpi_{\mathcal{S}}$ is indeed a closure operator.

A static process ϖ can then be equivalently characterized by a set of flows (that denote executions of the system) or by the set of valuations $[\![\varpi]\!]_{\mathcal{V}_A}$ (with no reference to their instant of occurrence: a valuation denotes a snapshot of the system). The counter given in Ex. 2.13 is not static since the value of a depends on the value it carried at the previous instant, while the merge given in Ex. 2.14 is static.

Clock Algebra Associated to a Valuation Since models of static processes are valuations more than traces, the language $\langle U, \cap, \cup, \setminus \rangle$ describing clocks presented in Sect. 2.2.4 should now be interpreted into a clock algebra with respect to a valuation $V \in \mathcal{V}_A$. Since a valuation represents a snapshot of execution, the local time referential it denotes contains at most one instant. Given $V \in \mathcal{V}_S$, the constant U is then interpreted as $\emptyset$ if $V = {}^{\star}V_A$, otherwise as a singleton we write $\{\cdot\}$. $\widehat{x} \in \mathcal{K}$ is interpreted as $\emptyset$ if $V(x) = \star$, otherwise as $\{\cdot\}$. The clock algebra associated to a valuation is then isomorphic to a boolean algebra with two elements.

2.5 Transition Systems as Models for Specifications

Numerous verification analyses apply to transition systems (e.g. [24, 6, 14]). Synchronous specifications can obviously be given a semantics by means of such transition systems, provided they can cope with the synchronous paradigm. Several models have been proposed: two of them are presented in this section. The main difference with the trace semantics is the formalization of the notion of the *internal state* of a process, via the introduction of *memory* or *state variables*. Traces correspond to the classical notion of runs of a transition system, but a process can be exactly described by a transition system only if properties of the parallel composition given on page 11 (in particular closure under bounded stuttering) are specified.

Synchronous Transition Systems (STS) are a very general model applied to the whole synchronous paradigm. They extend the classical notion of transition systems with variables that can be absent or present (domains are extended with the special value $\star$) and communication as synchronous composition. Persistent variables (often called memory variables) are distinguished from volatile ones. STS are extended to *Fair* STS in [47]: fairness conditions give a semantics close to the trace semantics, since stuttering with silent moves is finite. In the absence of fairness conditions, STS are rather used to give an operational semantics of languages: closure under bounded stuttering is not ensured. Such a semantics is clearly adopted in e.g. some analyses dedicated to SIGNAL [38] and [11], that can be described in terms of *Symbolic Labeled Transition Systems* (SLTS). An SLTS is like a STS a classical transition system extended with $\star$. Its originality in relation to STS relies on a clear distinction between the static (labels) and the dynamic (transition relation) parts of a process. Moreover the distinction between memory variables and others is made clearer: for these reasons SLTS will by preferred to STS in the following. Apart from these syntactical differences, STS and SLTS have the same expressiveness. STS and FSTS are presented in Sect. 2.5.1, SLTS are presented in Sect. 2.5.2. In both cases examples are given and the link with the trace semantics is discussed.

2.5.1 Synchronous Transition Systems (STS)

We first give notations for STS followed by an example, then present FSTS.

Plain STS Let us first present the terminology used for STS. Given a finite set of (typed) variables S (whose domain is extended with $\star$), a *state* is a (type-consistent) interpretation of variables (then states should be thought as *valuations* in the sense of the trace semantics). The set of states is denoted by $\mathcal{S}_S$. An STS is a tuple

$$\Theta : \langle S, \theta, \rho \rangle$$

where $\theta(S)$ is an assertion over variables that characterizes the set of *initial states*; and $\rho(S, S')$ is the *transition relation*, that is an assertion relating the current state s to the next one s' (classically, primed (resp. unprimed) variables refer to values in the next (resp. current) state). Given an assertion φ over variables $S \cup S'$ and states $s_i, s_j \in \mathcal{S}_S$, $(s_i, s_j) \models \varphi(S, S')$ means that $\varphi(s_i[S], s_j[S'])$ is true. A *run* (or a *trajectory*) is classically an infinite sequence of states $s_0 s_1 s_2 \ldots$ in $\mathcal{S}_S$ such that

$$s_0 \models \theta \text{ and } \forall i \in \mathbb{N}, (s_i, s_{i+1}) \models \rho$$

The composition of two STS $\Theta_1 : \langle S_1, \theta_1, \rho_1 \rangle$ and $\Theta_2 : \langle S_2, \theta_2, \rho_2 \rangle$ is given by:

$$\Theta_1 \parallel \Theta_2 : \langle S_1 \cup S_2, \theta_1 \wedge \theta_2, \rho_1 \wedge \rho_2 \rangle$$

The modeling of data-flow specifications into STS distinguishes two kinds of variables:

- the *reactive* or *volatile* variables are initial variables (signals) of the specification: they are absent or present at a given instant;

- some auxiliary variables (called *persistent* or *memory* variables) are used to memorize the value of some volatile variables from an instant to the next one. They represent the *internal state* of the process. Initially they are absent and can stay absent, nevertheless they remain always present after their first instant of presence.

Using notations of [47], the auxiliary memory variable $x.a$ memorizes the value of the volatile variable a by:

$$\theta \textbf{ implies } x.a = \star \ , \ x'.a = \textbf{if } a' \neq \star \textbf{ then } a' \textbf{ else } x.a$$

meaning that if in state s' a is present then $x.a$ memorizes its value, while if a is absent $x.a$ remains unchanged. It ensures that $x.a$ carries the value that a took at its last instant of presence.

Example 2.15 Let us consider again the counter given in Ex. 2.13. Since the volatile variable ma takes the previous value of the volatile variable a, an auxiliary memory variable $x.a$ is needed. If ma is present in the state s_{i+1} then either it was never present before (indicated by $x.a = \star$ in state s_i) then takes the initial value 0, or it takes the value $x.a$ had in state s_i. a takes the value of N when N is present i.e. when $ma = 0$, else it decrements; i.e. takes the value $ma - 1$. Moreover a and ma are synchronous. We then obtain the STS $\Theta = \langle S, \theta, \rho \rangle$ where

$$S = \{a, ma, N, x.a\} \ , \ \theta(S) : (a = \star \wedge ma = \star \wedge N = \star \wedge x.a = \star)$$

The transition relation ρ is given on Fig. 10(a) in the style of [46]. A run for Θ is given on Fig. 10(b): note that this infinite succession of states is reminiscent of a trace. ◇

$$\rho(S, S') : \begin{cases} a' = \begin{cases} \textbf{if } N' \neq \star \textbf{ then } N' \\ \textbf{else if } ma' \neq \star \textbf{ then } ma' - 1 \textbf{ else } \star \end{cases} \\ \wedge\ x'.a = \textbf{if } a' \neq \star \textbf{ then } a' \textbf{ else } x.a \\ \wedge\ ma' = \begin{cases} \textbf{if } a' = \star \textbf{ then } \star \\ \textbf{else if } x.a = \star \textbf{ then } 0 \textbf{ else } x.a \end{cases} \\ \wedge\ ma' = 0 \Leftrightarrow N' \neq \star \Leftrightarrow N' \geq 0 \end{cases}$$

(a) Transition relation

N	$\star$	1	$\star$	$\star$	0	4	$\star$	$\star$	...
a	$\star$	1	$\star$	0	0	4	3	2	...
ma	$\star$	0	$\star$	1	0	0	4	3	...
$x.a$	$\star$	1	1	0	0	4	3	2	...

(b) A run

Fig. 10: STS Θ of Ex.2.15

STS are very general and express the essential of the synchronous paradigm. As far as the state-based formalism is concerned, they e.g. have been used to formalize the StateMate semantics of Statecharts [17]. The model has been applied to the data-flow formalism, in particular to Signal. [57] uses them to describe the construction of hierarchic normal forms for transition relations. A slight variant is used in [46] to formally validate the translation from Signal specifications to C code performed by the Signal compiler.

The restriction of a trajectory to volatile variables is clearly related to a trace of the initial specification, in the sense of the trace semantics (Sect. 2.2.2). However note that so far STS express neither that processes are closed under stuttering, nor that the parallel composition between processes is fair. However FSTS are much closer to the trace semantics, as explained below.

Fair STS (FSTS) FSTS are presented in [47]. In order to cope with the complex Statecharts semantics and independently of the fairness issue, the set of variables S used in STS is refined into a disjoint union $S = S_L \cup S_E$, where *local* variables in S_L are distinguished from *externally observable* ones in S_E. Among variables in S_E, the set of *synchronization* variables $S_S \subseteq S_E$ is used by the process to synchronize with its environment (as opposed to the set of *controllable* variables $S_C = S \setminus S_S$).

[47] focuses on *realizable* STS, i.e. such that every state s in a run has a successor by ρ in which all synchronization variables in S_S are absent. This means that the system can wait for the environment to be ready for synchronization: it is a particular case of stuttering. General stuttering in the sense of the trace semantics (see Sect. 2.3.2: processes are set of traces closed under stuttering) is also dealt with. A state s_1 is a *stuttering variant* of a state s if all volatile variables are set to $\star$ and s_1 and s agree on persistent variables: the system remains in the same "memory" state. FSTS *closed under stuttering* are then presented.

FSTS extend STS with *fairness* conditions. A controllable signal variable $x \in S_C$ is said to be *enabled* with respect to S_S in the state s of a run if there exists a successor of s in which

$$x \neq \star \text{ and } \forall y \in S_S, y = \star.$$

Intuitively, x is enabled to carry a non-$\star$ value without the need of a synchronization with the environment, which means that the process can act significantly according to its internal state. The *justice requirement* states that if any controllable signal is continuously enabled from a given instant in a run, then it will eventually carry a non-$\star$ value. It is a generalization to STS of the fact that infinite stuttering can only come from an impossible synchronization. [47] gives an extension of the linear temporal logic LTL and deductive rules which allow the specification and proof of liveness properties.

2.5.2 Symbolic Labeled Transition Systems (sLTS)

Like STS, sLTS formalize the notion of internal state of a process by memory variables. Nevertheless they are clearly separated from classical variables that can be absent[6]. Memories are really persistent variables, and not particular variables that are initially absent then always present like in STS.

A sLTS Θ is denoted by

$$\Theta = \langle S, \xi, \theta, \rho, \mathcal{C} \rangle$$

where ξ is a set of memory variables and S a set of (signal) variables. *States*[7] of Θ are valuations on ξ and *transitions* are labeled by valuations on S. Formally:

- $\theta(\xi)$ is the *initialization predicate* (over variables ξ);
- $\rho(\xi \cup \xi' \cup S)$ is the transition relation (over variables $\xi \cup \xi' \cup S$); it characterizes the *dynamic* of the system: the evolution of its memories, the change of its internal state;
- the constraint $\mathcal{C}(S)$ denotes an invariant property between signal variables. Its set of solutions is a subset of $\mathcal{V}_S$, called the set of *admissible valuations*, and is the set of labels of Θ. $\mathcal{C}(S)$ is an upper-approximation of the set of valuations that actually occur in an execution, when only reachable states are considered. $\mathcal{C}(S)$ then characterizes the *static* aspects of the system.

A process $\varpi \in \mathcal{P}_S$ can be associated to a sLTS $\Theta = \langle S, \xi, \theta, \rho, \mathcal{C} \rangle$, like illustrated in Ex. 2.16. Admissible valuations are then an upper-approximation of $[\![\varpi]\!]_{\mathcal{V}_A}$ (see Sect. 2.4.3). In the following the memory variable associated to the signal variable $x \in S$ is denoted by ξ_x.

Example 2.16 Let us give an sLTS Θ for the counter presented in Ex. 2.13. A memory variable ξ_a is used to memorize the past value of a, as $x.a$ does in the STS. Initially, $\xi_a = 0$. Then at each action of the system, if a is present then its value is memorized in ξ_a which is updated, otherwise ξ_a does not change. Moreover if ma is present it takes the value of ξ_a. We then have $\xi = \{\xi_a\}$ and the evolution of the internal state of the counter (its dynamic part) is described by:

$$\theta(\xi_a) : (\xi_a = 0) \tag{3}$$

$$\rho(\xi_a, \xi'_a, ma, a) : \begin{cases} \xi'_a = \begin{cases} a \textbf{ if } a \neq \star \\ \xi_a \textbf{ else} \end{cases} \\ \textbf{if } ma \neq \star \textbf{ then } ma = \xi_a \end{cases} \tag{4}$$

Let us discuss now the static part of Θ, i.e. the relations that hold between signal variables in any action. Similarly to Ex. 2.15 we obtain:

$$\mathcal{C}(ma, a) : \begin{cases} & N = \star \wedge a = \star \wedge ma = \star \\ \bigvee & N \neq \star \wedge ma = 0 \wedge a = N \wedge N \geq 0 \\ \bigvee & N = \star \wedge ma \neq \star \wedge ma \neq 0 \wedge a = ma - 1 \end{cases} \tag{5}$$

Equations (3), (4) and (5) describe Θ, which can be equivalently described by the automaton given on Fig. 11. Note the stuttering transitions labeled by a silent valuation that loop on each state. A trace of execution of Θ is also given on Fig. 12 which fires all non-stuttering transitions. We explicit the values of state variables (they are traditionally left implicit since redundant with values of ma). ◇

[6] the difference is clearly marked in e.g. [38] by the use of terms "memory variables" as opposed to "signal variables".

[7] sLTS and STS use the same terminology to denote different objects: for STS the term "state" denotes a valuation on the set of persistent *and* volatile variables.

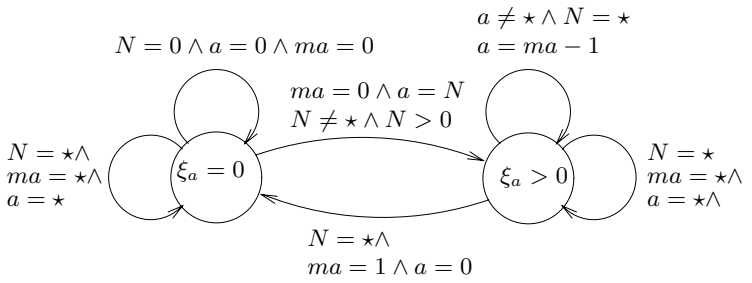

Fig. 11: An automaton for Ex.2.16

N		1		$\star$		0		4		$\star$		$\star$	...
a		1		0		0		4		3		2	...
ma		0		1		0		0		4		3	...
ξ_a	0	$\longrightarrow$	1	$\longrightarrow$	0	$\longrightarrow$	0	$\longrightarrow$	4	$\longrightarrow$	3	$\longrightarrow$	...

Fig. 12: A trace of execution for Ex.2.16

Example 2.16 highlights several differences between SLTS and STS. SLTS make a clear separation between the static and the dynamic part of a process. The dynamic part addresses the evolution of memory variables: unlike STS the initialization predicate only concerns memory variables and a primed signal variable in the transition relation has no meaning. As a result, a *run* is an infinite sequence of memory states, decorated by the infinite sequence of the valuations for signals variables that triggered the evolution from one memory state to the other. There is no "vertical" representation like for STS.

The restriction of a run of Θ to variables in S is nevertheless related to a trace of the associated process, with a weaker meaning than the trace semantics, as for STS with no fairness conditions. Note that it should be possible to meet the trace semantics by considering a pair composed of a transition system and a fairness hypothesis.

3 Specification and verification using SIGNAL

In the following SIGNAL [45, 25, 9] is used as a description language for data-flow specifications. Notations given in Sect. 2 are still valid. The kernel of SIGNAL is first presented in Sect. 3.1, then two analyses that model a SIGNAL specification by a symbolic transition system are described in Sect. 3.2.

3.1 The Kernel of SIGNAL

A SIGNAL *program* or *specification*[8] on a set of variables A is written $\mathtt{P} \in \mathcal{P}g_A$. **It denotes semantically a set of flows** called a SIGNAL process (then a subset of $\mathcal{F}_A$) written $[\![\mathtt{P}]\!]$ or $\varpi_{\mathtt{P}}$. $1_{\mathcal{P}g_A} \in \mathcal{P}g_A$ denotes a program that does not constrain variables in A (it can be thought as a program containing no equations): $[\![1_{\mathcal{P}g_A}]\!] = \mathcal{F}_A$.

The kernel language is composed of four elementary equation patterns that denote *elementary processes*. The instantaneous function and the delay presented in Sect. 3.1.1 are monochronous, while the filtering and the deterministic merge presented in Sect. 3.1.2 are polychronous. More complex programs are built using the parallel composition operator |, which corresponds to the parallel composition of processes presented in Sect. 2.3.1:

$$[\![\mathtt{P}_1 \mid \mathtt{P}_2]\!] \triangleq [\![\mathtt{P}_1]\!] \parallel [\![\mathtt{P}_2]\!]$$

The semantics of an elementary equation P is given as follows: $[\![\mathtt{P}]\!] = \{F \in \mathcal{F}_A \mid \forall t \in \mathbb{N},\ exp(F,t)\}$ (where $exp(F,t)$ is an expression built on F and t) is written more concisely $\mathtt{P} \rightsquigarrow exp(F,t)$. To simplify this expression, $F(a)(t)$ is written a_t^F or a_t for a variable $a \in A$, whose domain is denoted by D_a.

[8] The term "program" could imply that the specification is executable, which is not necessarily the case.

3.1.1 Monochronous Operators

As said in Sect. 2.4, a process ϖ on $A = \{a_i\}_{i=1...n}$ is monochronous if all variables in A are synchronous. Then $F \in \varpi$ implies that for any $i, j \in [1, n]$: $\forall t, {a_i}_t^F = \star \Leftrightarrow {a_j}_t^F = \star$. This constraint appears in the semantics of the two monochronous operators *instantaneous function* and *delay*.

Instantaneous Function Let $y, x_1, \ldots, x_n$ be variables (not necessary distinct) of domain D_y and D_{x_i} respectively, A be the set $\{y, x_1, \ldots, x_n\}$ and g be a function from $D_{x_1} \times \cdots \times D_{x_n}$ to D_y, such as addition for instance. Such functions are extended to signals in a synchronous way: the function is applied iff all its arguments are present; then the evaluation of an expression like $\star + 2$ is irrelevant. The general semantics is:

$$[\![\texttt{y := g(x}_1\texttt{, ... ,x}_n\texttt{)}]\!] \rightsquigarrow \begin{array}{l} y_t = \star \Rightarrow x_{1_t} = \cdots = x_{n_t} = \star \\ \bigwedge \quad y_t \neq \star \Rightarrow \ \forall i, x_{i_t} \neq \star \wedge y_t = g(x_{1_t}, \ldots, x_{n_t}) \end{array}$$

When y is present it takes the value $g(x_1, \ldots, x_n)$. Functions allow the construction of monochronous predicates (e.g. `c := x<y` builds the predicate $x < y$; and c, x and y are always both absent or present at a given instant). An equation can also describe a *relation* between variables: the process `y := x or y` on $\{y, x\}$ states that when y and x are present, their values verify $x_t \Rightarrow y_t$.

Delay The *delay* operator `$` aims at defining the dynamic of systems (see Sect. 2.5.2). Let x, y be variables (not necessarily distinct) such that $D_x = D_y$, and let $v_0 \in D_y$. The elementary process on $\{x, y\}$ (or *delay equation*):

$$[\![\texttt{y := x \$1 init v0}]\!] \rightsquigarrow \begin{array}{l} y_t = \star \Leftrightarrow x_t = \star \\ \bigwedge \quad (x_0 \neq \star \Rightarrow y_0 = v_0) \ \bigwedge \ (t \geq 1 \Rightarrow y_t = x_{t-1}) \end{array}$$

defines the variable y as the *delay* of the (*delayed*) variable x. y and x are synchronous and when present, y takes the previous/delayed non-$\star$ value of x, initially v_0.

Remark 3.1 One can wonder why the flow semantics of the delay is expressed by the sub-expression $x_0 \neq \star \Rightarrow y_0 = v_0$, and not by the one $y_0 = v_0$ which seems to express exactly an initialization at the first instant. However the silent flow must belong to the process : in this case the initial value v_0 is never carried by y. Note also that the trace semantics of the delay is far less easy to write than the flow one. Indeed, it must be expressed that y carries the last non-$\star$ value of x in case of presence. Let us consider a flow $F \in \mathcal{F}_{\{y,x\}}$ such that $F \in [\![$ `y := x $1 init v0` $]\!]$. Then for all $t \in \mathbb{N}$: $y_t^F \neq \star \Leftrightarrow x_t^F \neq \star$. Assume $y_t^F \neq \star$ for $t > 0$, then $F(t) \neq {}^\star V$ and by definition of flows, $F(t-1) \neq {}^\star V$. Therefore $y_{t-1}^F \neq \star$ and $x_{t-1}^F \neq \star$: if $y_t^F \neq \star$, the last non-$\star$ value of x is x_{t-1}^F. On the contrary a trace T does not have this property of flows, and the search for the last non-$\star$ value of x must be made explicit by:

$$y_t^T \neq \star \wedge t' = max\{s.\ s < t \wedge y_s^T \neq \star\} \ \Rightarrow \ y_t^T = x_{t'}^T$$

Similarly the initial valuation must be expressed by: $t' = min\{t.\ y_t^T \neq \star\} \ \Rightarrow \ y_{t'}^T = v_0$. ◇

If we reuse notations of Sect 2.5.2 the sLTS associated to the above delay equation is such that:

$$\theta(\xi_x) : (\xi_x = v_0) \quad , \quad \mathcal{C}(y, x) : (y = \star \Leftrightarrow x = \star) \quad , \quad \rho(\xi_x, \xi'_x, y, x) : \left\{ \begin{array}{l} \xi'_x = \left\{ \begin{array}{l} x \textbf{ if } x \neq \star \\ \xi_x \textbf{ else} \end{array} \right. \\ \textbf{if } y \neq \star \textbf{ then } y = \xi_x \end{array} \right. \tag{6}$$

Note that $\mathcal{C}(S)$ states a clock equivalence between x and y that are synchronous.

Remark 3.2 The delay is used to represent SIGNAL constants: the constant `v` is materialized by a signal $\texttt{z}_\texttt{v}$ which carries `v` at each of its instants of presence. The constant `v` is by this way given a clock, which is $\widehat{z_v}$ and is not constrained by the equation. E.g. for `v` equal to 2: $\texttt{z}_2$ `:=` $\texttt{z}_2$ `$ 1 init 2`. ◇

3.1.2 Polychronous Operators

The filtering and the deterministic merge operators are polychronous: the involved variables are not necessarily synchronous nevertheless their clocks are constrained to verify a given relation, as shown below.

Filtering Let x, y be variables (not necessarily distinct) such that $D_x = D_y$, and c be a boolean variable. The elementary process defined on $\{x, y, c\}$

$$[\![\ \texttt{y := x when c}\ ]\!] \rightsquigarrow \bigwedge \begin{array}{l} c_t \neq true \Rightarrow y_t = \star \\ c_t = true \Rightarrow y_t = x_t \end{array}$$

states that if c is present and *true* then y takes the value of x, otherwise y is absent. The `when` operator performs a filtering or an under-sampling: it extracts a sub-clock $\widehat{y}$ from $\widehat{x}$ according to the truth values of the predicate c. This predicate is called the *guard* of the filtering. The `when` operator constrains the clock of x and y as follows: $\widehat{y}$ is equal to the intersection of $\widehat{x}$ with instants where c is present and *true*.

Deterministic Merge Let y, u, v be variables (not necessarily distinct) such that $D_y = D_u = D_v$. The elementary process defined on $\{y, u, v\}$

$$[\![\ \texttt{y := u default v}\ ]\!] \rightsquigarrow \bigwedge \begin{array}{l} u_t \neq \star \Rightarrow y_t = u_t \\ u_t = \star \Rightarrow y_t = v_t \end{array}$$

states that if u is present then y takes the value of u, otherwise the one of v. It corresponds to the merge with priority given to the first argument as in Ex. 2.14. The `default` operator constrains the clocks of y, u and v as follows: $\widehat{y}$ is equal to $\widehat{u} \cup \widehat{v}$.

3.1.3 Other Constructions

The complete language offers also the *hiding* of variables. `(| P |)/x` hides `x` in `P`; it corresponds to a projection of the process $[\![\texttt{P}]\!]$ on x making x a local variable:

$$\text{for } \texttt{P} \in \mathcal{P}g_A,\ [\![(\texttt{P / x})\ ]\!] \triangleq \prod_{A\setminus\{x\}}([\![\texttt{P}]\!])$$

Hiding is necessary when constants are involved, since as explained in Rem. 3.2 a constant `v` is represented by a variable $\texttt{z}_\texttt{v}$ which is purely local to the use of `v`. For example

`y := x + 2` stands for `(| y := x + z2 | z2 := z2 $ 1 init 2 |) / z2`.

Note that the clock of z_2 is now clearly fixed since the instantaneous function + constrains z_2 to be synchronous with x and y.

The language provides powerful constructions for modular programming we shall not deal with here. We just mention two very useful *derived operators*:

- The operator `^=` constrains its arguments to be synchronous:

  $$[\![\ \texttt{x ^= y}\ ]\!] = \{F \in \mathcal{F}_{\{x,y\}} \mid x_t^F = \star \Leftrightarrow y_t^F = \star\}$$

  `x ^= y` stands for `(| b1 := when x=x | b2 := when y=y | b := b1 = b2 |)/ b,b1,b2`.

- `event x` stands for the boolean expression `x=x`. It denotes a pure event (of type $\{\star, true\}$) synchronous to x, that is $\widehat{x}$.

Example 3.1 The counter described in Ex. 2.13 and 2.16 can be specified by the following system of equations, where the last equation guarantees the determinism of the process. Note that the third equation specifies a constraint on the input N, which could be expressed by the LUSTRE expression `assert(N>=0)`.

```
(| ma := a $1 init 1
 | a := N default ma-1
 | N ^= when N>=0
 | N ^= when ma=0 |)
```

◇

Example 3.2 The example of Fig. 13 shows a SIGNAL program that computes an absolute value and one of its flows. The program defines the *model of process* `abs` (keyword `process`) on $\{y, a\}$ whose integer input variable (indicated by `?`) is `y`, whose integer output variable (indicated by `!`) is `a` and whose integer local variables `p` and `n` are declared by the keyword `where`. The system of equations assigns to `p` (resp. `n`) the value of `y` (resp. `-y`) at all instants s.t. `y` is positive (resp. strictly negative). These two variables are *exclusive* (they cannot be both present at the same time). Then `a` is assigned the value of `p` if `p` is present, else the value of `n`, so `a` is the absolute value of `y`.

◇

```
process abs =
  (? integer y;
   ! integer a)
  (| p : = y when y>=0
   | n := -y when y<0
   | a :=  p default n
   |) where integer p,n end
```

	t_0	t_1	t_2	t_3	t_4	...
y	3	4	-2	0	-5	...
p	3	4	$\star$	0	$\star$	...
n	$\star$	$\star$	2	$\star$	5	...
a	3	4	2	0	5	...

Fig. 13: An absolute value in SIGNAL

3.2 Analyses on SIGNAL

POLYCHRONY proposes a range of tools dedicated to particular analyses applied to specifications. Two analyses are presented here in a concise way, which both consider a process as a symbolic transition system (the first one uses explicitly SLTS). An analysis [38] of the trajectories of a boolean system in described is Sect. 3.2.1 and an analysis [11] on non-boolean values is described in Sect. 3.2.2.

3.2.1 Boolean Analysis of Trajectories

The analysis [38] implemented in the tool SIGALI applies to boolean specifications. It uses systems of polynomial dynamic equations over the field $\mathbb{Z}/3\mathbb{Z}$ as a description of a SLTS. The principle is to encode the three possible status of a boolean variable y into a signal variable Y, with the following meaning:

Y		y
+1	$\mapsto$	*present* and *true*
-1	$\mapsto$	*present* and *false*
0	$\mapsto$	*absent*

E.g. for boolean signals a, b, c the equation `c := a default b` is denoted by the polynomial equation $C = A+(1-A^2)B$ which represents exactly the admissible valuations of the equation. For non-boolean variables, the equation is represented by the polynomial $C^2 = A^2+B^2-A^2B^2$, which only keeps track of the synchronization constraint $\widehat{c} = \widehat{a} \cup \widehat{b}$. The polynomial dynamic system is formally reorganized into three sub-systems as in Sect. 2.5.1, where Q_0, P and Q are polynomials such that:

- $Q_0(\xi) = 0$ corresponds to the initialization predicate θ;
- $\xi' = P(\xi, S)$ corresponds to the transition relation ρ, except that the value taken by the delay variable y in `y := x $ init v0` (if $y \neq \star$ then $y = \xi_x$ in Eq. (6)) is specified in Q;
- $Q(\xi, S) = 0$ denotes the admissible valuations of the system in a given state.

Note that it is possible to specify that a signal variable x is always present via the polynomial equation $X^2-1 = 0$, what is out of the question in the trace semantics.

SIGALI symbolically handles polynomial equations: they are efficiently implemented by TDD (Ternary Decision Diagram) which are BDD (Binary Decision Diagram [51]) whose edges are labeled by 0, 1 but also -1. SIGALI provides analyses of the trajectories of systems: symbolic model-checking (e.g. safety, liveness properties) and controller synthesis [41, 40].

3.2.2 Non-boolean Values Handling

The analysis [11] (not integrated to POLYCHRONY) considers symbolic transition systems that contain linear relations between values of the non-boolean variables of a specification. In practice it uses a polyhedral encoding inspired from [30] and extended to consider the special value $\star$. Firstly, an upper-approximation of the set of admissible valuations (called "labels") is computed, and used to represent the transition relation ρ. Secondly, ρ is used to compute an upper-approximation of the set of reachable states, and a widening operator is proposed. We focus here on the implementation of the first part.

A set of valuations for a program P is symbolically represented by a set of constraints sets, following directly the intuition given in Ex. 2.16. An upper-approximation of the set of admissible valuations is extracted from P, using deductive rules that express a behavioral semantics of the kernel of SIGNAL. Practically, a set of

constraints sets is first associated to each equation. E.g. the equation `y := x when c` is encoded by the set $C = \{C_1, C_2, C_3\}$ where

$$C_1 : \{y = x, c = \mathit{true}\},\ C_2 : \{y = \star, c = \mathit{false}\},\ C_3 : \{y = \star, c = \star\}.$$

Each C_i denotes a set of valuations and C denotes their union. Then equations are put in parallel: valuations associated to single equations are composed to obtain all possible configurations of a parallel composition. For example:

$$\textbf{if } \texttt{eq1} : \{C_1, C_2\} \text{ and } \texttt{eq2} : \{C_3, C_4\} \textbf{ then } \texttt{eq1 | eq2} : \{C_1 \cup C_3, C_1 \cup C_4, C_2 \cup C_3, C_2 \cup C_4\}$$

A naive computation of the resulting set would yield an exponential number of constraints sets, among which unsatisfiable constraints should be discarded. A set of constraints can be unsatisfiable for two reasons:

- a variable is constrained to be both present and absent, e.g. $\{x = \star, x = v\} \subseteq C_i$, where $v \in D$;
- a variable is present but the constraint on the value it carries has no solution, e.g. $\{x \neq \star, x > 3, x < 0\} \subseteq C_i$.

In practice, constraints sets are *normalized* and split according to the status of variables. As a result, any normalised constraints set is implemented by a convex polyhedron stating constraints on present variables (and memory variables) plus a *list of absent variables*.

Example 3.3 Let us consider Ex. 3.2. To the equation `p := y when y>=0` is associated the set $C^p = \{C_1^p, C_2^p, C_3^p\}$ where

$$C_1^p : \{p = y, y \geq 0\},\ C_2^p : \{p = \star, y < 0\},\ C_3^p : \{p = \star, y = \star\}.$$

To the equation `n := -y when y<0` is associated the set $C^n = \{C_1^n, C_2^n, C_3^n\}$ where

$$C_1^n : \{n = -y, y < 0\},\ C_2^n : \{n = \star, y \geq 0\},\ C_3^n : \{n = \star, y = \star\}.$$

The parallel composition of these two equations produces a set of 9 sets of contraints among which only 3 are kept:

$$C_1^p \cup C_2^n : \{p = y, y \geq 0, n = \star\},\ C_2^p \cup C_1^n : \{p = \star, y < 0, n = -y\},\ C_3^p \cup C_3^n : \{p = \star, y = \star, n = \star\}.$$

The other ones are discarded either because one variable is constrained to be both present and absent (e.g. y in $C_1^p \cup C_3^n$), or because the constraints $y < 0$ and $y \geq 0$ appear in the set (e.g. in $C_2^p \cup C_1^n$). Finally the normalization of constraints sets gives the following results:

$$C_1 \ :\ \{(y = p = n = a = \star)\}\ ,\ C_2 \ :\ \{(n = \star), (y \geq 0 \wedge p = y = a)\}\ ,\ C_3 \ :\ \{(p = \star), (y < 0 \wedge n = -y = a)\}$$

◇

4 Abstraction-based Analyses

Abstractions are very classically applied to specifications to facilitate their verification. Specifications are often composed of a control and a data part: the state-space to be explored can be huge or even infinite due to the data part. Abstractions aim at reducing this state-space, or even to fall in a finite one. Various aspects of abstractions have been studied: mainly correctness (a *safe* abstraction ensures that some properties verified by the abstract specification are also verified by the concrete one, see e.g. [20]) and termination of analyses (see [20] or e.g. [12]), compositionality (decomposition of problems into smaller independent ones, see e.g. [6]), etc. A widely used theoretical framework to design abstractions and analyses, called *abstract interpretation*, is presented in Sect. 4.1. Abstraction means abstract domains for computations: standard ones are given in Sect. 4.2.

4.1 Notions of Abstract Interpretation

According to [19], "abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems, e.g. computations of programs. In particular program analysis algorithms can be constructively derived from these abstract semantics". We give an informal presentation in Sect. 4.1.1 then present the formal framework in Sect. 4.1.2.

4.1.1 Informal Presentation

Static analysis[9] aims at obtaining as much information and properties as possible about the possible runs of a program without actually having to run it on all input data. It is fully automatic and based on *non-standard executions* that perform the computations of the program using a description of values (or *abstract values*), instead of the actual computed values. *Abstract interpretation* is a theory that expresses static analysis as a formal correspondence between the concrete semantics of programs and an abstract semantics guided by the property of interest. Most of the work aims at finding a *safe* description of the behavior of programs (by abstract computations in abstract domains), i.e. such that the result of the analysis can always be depended on: the answer can be "yes", "no" or "I don't know". Pionner work of Cousot and Cousot [20] strongly influenced following works on abstract interpretation since it defines a unified framework for a number of program analyses. A general and more intuitive introduction can be found in [36].

Given a program to be analyzed, the property under interest induces the choice of a *concrete domain* Conc and of an *abstract domain* Abs. In an example given by [36], the problem is to determine whether the values of some integer variable at some control point are all even, all odd, or both of them. Then the concrete domain is $(\mathcal{P}(\mathbb{N}), \subseteq, \cap, \cup)$ and the abstract domain $(\mathsf{Abs}, \sqsubseteq_a, \sqcap_a, \sqcup_a)$ is such that $\mathsf{Abs} = \{\bot, \text{even}, \text{odd}, \top\}$ is endowed with the partial order $\bot \sqsubseteq_a \{\text{even}, \text{odd}\} \sqsubseteq_a \top$. Abs and Conc are connected by an *abstraction function* $\alpha : \mathsf{Conc} \to \mathsf{Abs}$, and a *concretization function* $\gamma : \mathsf{Abs} \to \mathsf{Conc}$. Any concrete function $\mathsf{op_c} : \mathsf{Conc} \to \mathsf{Conc}$ (e.g. addition) is modeled in Abs by an abstract function $\mathsf{op_a} : \mathsf{Abs} \to \mathsf{Abs}$ (e.g. a function that indicates the parity of the result of an addition depending on the parity of its arguments) in such a way that any operation $\mathsf{op_c}$ on $x_c \in \mathsf{Conc}$ is imitated by its abstraction $\mathsf{op_a}(\alpha(x_c))$. Finally a set $acc \in \mathsf{Conc}$ (resp. $abs \in \mathsf{Abs}$) of concrete (resp. abstract) reachable states is associated to each control point, resulting in a system of recursive equations which remains to be solved (e.g. the set of states end reached at the end of the body of a loop is expressed in function of the set of states begin reached at the beginning of the body, itself expressed in function on end and on the condition of the loop).

4.1.2 Theoretical Foundations

The ideas informally presented in Sect. 4.1.1 are formalized in [20]. Very general solutions are proposed, and illustrated on a simple flow-chart language.

Definition of the abstraction Abstract and concrete domains are *complete lattices* $(\mathsf{Abs}, \sqsubseteq_a, \sqcup_a, \sqcap_a)$ resp. $(\mathsf{Conc}, \sqsubseteq_c, \sqcup_c, \sqcap_c)$. The pair (α, γ) is a *Galois insertion*, that is it verifies the following properties:

1. α and γ are monotonous, e.g. for all $x_c^1, x_c^2 \in \mathsf{Conc}$, $x_c^1 \sqsubseteq_c x_c^2$ implies $\alpha(x_c^1) \sqsubseteq_a \alpha(x_c^2)$;
2. $\forall x_a \in \mathsf{Abs}$, $x_a = \alpha \circ \gamma(x_a)$ (a *Galois connection* must only ensure that $\alpha \circ \gamma(x_a) \sqsubseteq_a x_a$);
3. $\forall x_c \in \mathsf{Conc}$, $x_c \sqsubseteq_c \gamma \circ \alpha(x_c)$.

Point (1) is intuitive: comparisons in the concrete resp. abstract domains must be preserved by abstraction and concretization. Point (2) means that concretization introduces no loss of information. Point (3) means as expected that abstraction introduces a loss of information: abstraction followed by concretization gives a set of configurations bigger than the original one.

A *consistent* (or *safe*, or *conservative*) abstraction of concrete functions can now be defined. Let $\mathsf{op_c} : \mathsf{Conc} \to \mathsf{Conc}$ be a concrete function and $\mathsf{op_a} : \mathsf{Abs} \to \mathsf{Abs}$ the corresponding abstract one. The abstract interpretation is safe if the result of an abstract operation at least contains the result of the concrete one:

$$\forall x_c \in \mathsf{Conc},\ \mathsf{op_c}(x_c) \ \sqsubseteq_c\ \gamma \circ \mathsf{op_a} \circ \alpha(x_c), \quad \text{equivalently } \alpha \circ \mathsf{op_c}(x_c) \ \sqsubseteq_a\ \mathsf{op_a} \circ \alpha(x_c)$$

which correspond to the diagrams shown on Fig. 14.

Accumulating Semantics The principle of *accumulating semantics* is to compute for each control point the set of possible reachable states (i.e. values for variables). Elementary constructions of the language are locally interpreted by order preserving functions, then the static semantics of programs is generally expressed as a set of recursive equations. The resolution is then a least fixpoint computation. On the one hand the Tarski fixpoint theorem ensures that its least fixpoint exists. On the other hand the choice of safe abstractions ensures that the least fixpoint in the abstract domain is an upper-approximation of the least fixpoint in the concrete one.

[9] not to be confused with analysis of a *static specification* in the sense of Sect. 2.4.3

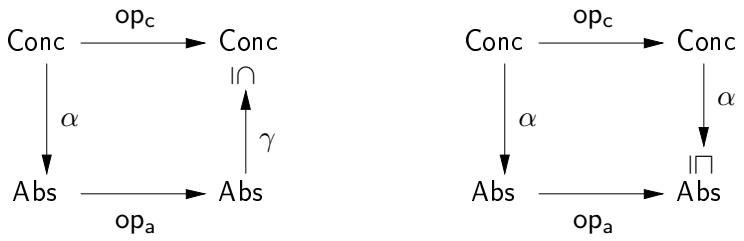

Fig. 14: Safe abstraction

Termination of the resolution An increasing iterative least fixpoint computation can yield to infinite computations if the abstract lattice contains infinite increasing chains (what is generally the case excepted for very restricted abstract domains). Assuming the iteration has the form

$$abs_0 = \bot_{\mathsf{Abs}} \quad , \quad abs_{i+1} = F_a(abs_i) \tag{7}$$

[20] proposes to over-approximate the limit of the sequence $(abs_n)_{0\leq n}$ using a *widening operator* $\nabla : \mathsf{Abs}\times\mathsf{Abs} \rightarrow \mathsf{Abs}$ such that:

1. $\forall x_a^1, x_a^2 \in \mathsf{Abs}$, $x_a^1 \sqcup_a x_a^2 \sqsubseteq_a x_a^1 \nabla x_a^2$
2. for all increasing sequence $(x_n)_{0\leq n}$ the sequence $(x'_n)_{0\leq n}$ defined by $x'_0 = x_0$, $x'_{i+1} = x'_i \nabla x_{i+1}$ is not strictly increasing.

The computation of Eq. (7) can then be replaced by the one

$$abs'_0 = \bot_{\mathsf{Abs}} \quad , \quad abs'_{i+1} = abs'_i \nabla F_a(abs'_i)$$

Applications Practically, abstract interpretation was mainly applied to the verification of invariance properties on a wide range of specification styles: transition systems [39], sequential, functional, logic, constraint logic, synchronous programming, etc. As far as synchronous programming is concerned, [27] explains how compiling specifications into interpreted automata and proving invariance properties by means of a synchronous observer are related to abstract interpretation. Moreover [35] expresses formally a clock analysis for LUSTRE and proves its correctness in the framework of abstract interpretation. It also emphasizes that abstraction of data-flow specifications can be formulated in the context of type systems.

4.2 Standard Abstract Domains

The precision of an abstraction depends on the choice of the abstract domain. The abstract state-space obtained by *boolean* abstractions presented in Sect. 4.2.1 is finite but abstractions are rather coarse. More expressive and possibly infinite abstract domains containing *non-boolean* values are shortly presented in Sect. 4.2.2.

4.2.1 Boolean Abstraction

This abstraction is well known in verification problematics, see e.g. the predicate abstraction used in [24, 6, 52]. One considers a set of k predicates φ_i on the variables of the concrete specification. A boolean variable B_i is associated to each φ_i: it represents all concrete states that satisfy the predicate φ_i. Abs can therefore be chosen as the set of all predicates on k boolean variables B_i. For exp (resp. φ) a concrete (resp. abstract) predicate, the most precise Galois insertion (α, γ) is then:

$$\gamma(\mathsf{exp}(B_1, \ldots, B_k)) = \mathsf{exp}[B_i \leftarrow \varphi_i]$$

$$\alpha(\varphi) = \bigwedge\{\mathsf{exp}(B_1, \ldots, B_k) \mid \varphi \Rightarrow \mathsf{exp}[B_i \leftarrow \varphi_i]\}$$

[24] notes that α is not easily computable in general. It chooses then for Abs the set of monomials on the boolean variables B_i (that is the set of conjunctions of B_i's and $\neg B_i$'s that contain each B_i at most once). α is then

$$\alpha(\varphi) = \bigwedge_{1\leq i\leq k} \{B_i \mid \varphi \Rightarrow \varphi_i\} \tag{8}$$

A particular choice for the abstraction predicates stands out, as pointed in [24, 52]. They consider dynamic systems specified in a Unity-like language where transitions are composed of *guards* and affectations. They show on practical examples that choosing guards for abstract predicates produces a relatively precise global control graph of the system. The precision of the abstraction is then sufficient to prove properties related to the control of the specification.

The development of *symbolic* techniques (e.g. symbolic model-checking [15]) has allowed to consider domains larger and larger with the use of BDDs (or Ordered BDDs for which numerous libraries exist), BEDs (Boolean Expression Diagrams [3, 58]), etc.

4.2.2 Non-boolean Abstractions

Well known abstract domains that contain non-boolean values are for example (roughly from the coarsest to the finest): intervals, regions, polyhedra, and formulas of the Presburger arithmetic (non-quantified fragment or full arithmetic). A few related analyses, data structures and tools are enumerated below, among others.

Intervals The abstract lattice of real intervals is well known: a widening operator was earlier defined in [20]. Intervals are used e.g. in the verification of timed automata [54] by means of IDDs, a data representation derived from BDDs [55]. Analyses based on intervals are not precise but also not costly: they can be applied as a first stage in the verification process.

Regions A set of constraints of the kind $x + c \leq y + d$ where $x, y \in \mathbb{R}$ and $c, d \in \mathbb{N}$ defines an area called a *region*, by allusion to regions described by real clocks when time evolves in timed automata (see e.g. [56]). Two data structures have been proposed to represent them: DDDs (Difference Decision Diagrams [43]) that describe set of constraints of the kind $x - y \leq c$ and similar CDDs (Clock Difference Diagrams [4]). Both belong to the family of *interpreted* BDDs, that refer to BDDs whose nodes are pseudo-variables that correspond to non-boolean constraints.

Convex Polyhedra A polyhedron determines a linear subspace of $\mathbb{Q}^n$ of the kind $\{X \in \mathbb{Q}^n \mid \bigwedge a_i.x_i \geq b_i\}$ for $a_i, a_j \in \mathbb{Q}$. It can be used to represent a subset of $\mathbb{R}$ or $\mathbb{Z}$. Polyhedra are very popular since they offer a good compromise between cost and accuracy. The lattice of convex polyhedra and a widening operator where firstly proposed in [21] and widely further used (see e.g. [31, 32]).

Presburger Formulas The Presburger algebra (or arithmetic) is a first order language that describes linear constraints between integer variables. It is more and more practically used in analyses as tools evolve. For example the tool Omega [48], based on polyhedral representations, was used e.g. in symbolic model-checking [13] or constraint-based array dependence analysis [49]. Presburger formulas can also be represented by automata [59]. Some algebraic properties of Presburger arithmetic are studied in [12], to e.g. restrict specifications to a fragment of the arithmetic so that analyses terminate (therefore there is no need to apply a widening).

4.2.3 Mixed Abstractions

Real specifications often contain several data-types, at least a boolean part that represents control and a non-boolean part that represents data. More and more verification techniques mix boolean and non-boolean abstract domains. Some of them use special data structures that mix interpreted and classical BDDs: IBDDs [22] for intervals, and for polyhedra DDCs (Decision Diagram with Constraints [42]) and more recent works of [34].

5 Abstractions and Analyses in POLYCHRONY

Very different kinds of abstraction are widely used in POLYCHRONY, described into very distinct formalisms. Authors of the work [11] claim its relation with abstract interpretation, and use two steps presented in Sect. 4.1.2: accumulating semantics and widening operator. On the contrary, some abstractions are described by means of syntactical transformations of SIGNAL programs. In these cases the correctness of abstractions is not explicitly expressed in terms of the underlying Galois insertion. Moreover analyses that address only static processes are not concerned with the accumulating semantics as defined by [20], thus neither with widening operators, though they produce a set of recursive equations. Finally, the particularly important abstraction by synchronizations is quite original in its principle, though it is only a particular case of boolean abstraction. For these reasons, the

whole set of studies involving abstractions in POLYCHRONY can only gain clarity from a general survey which would establish classifications and comparisons both between classes and with classical abstractions.

A data-flow process can be seen as the parallel composition of two or more components (which are also processes) that play a specific role in the system and have particular properties. Several decompositions into pairs of components are classically used in POLYCHRONY: one distinguishes in a process its *dynamic* part (which makes memories evolve once values for signal variables are computed) composed with its *static* or *combinatorial* part (responsible for the computation of these values for signal variables); its *synchronization* part (which describes relations between clocks of variables) composed with its *computational* part (which deals with values of variables); or its *boolean* part (which describes the *control* of the process) composed with a non-boolean part (which describes the processing of *data*). Decompositions are orthogonal and their composition leads to decompositions into tuples of components (for example decomposition of the data part into its static and dynamic parts). Such structural decompositions induce *structural abstractions*, meaning that a component of a process is considered alone as abstracting the whole process.

Historically, POLYCHRONY has first proposed methods and tools that rely on structural abstractions, presented in Sect. 5.1. As shown in Sect. 5.2, these abstractions are boolean ones. But classical abstractions into specific data domains start out: the analysis of [11] uses polyhedra and an analysis on intervals is in course of design. These kind of abstractions are presented in Sect. 5.4. All along the section, the way absence is handled is emphasized.

5.1 Structural Abstractions

The principle of structural abstractions is presented in Sect. 5.1.1, then the main abstractions and their compositions used in POLYCHRONY are given: abstraction by control in Sect. 5.1.2, static abstraction in Sect. 5.1.3, abstraction by synchronizations in Sect. 5.1.4 and composed abstractions in Sect. 5.1.5.

5.1.1 Principle of Structural Abstractions

Let us first recall the principle of the *synchronous observer*, widely used in the verification process of LUSTRE programs [29]. To determine whether a LUSTRE program $\mathtt{P}$ satisfies a safety property $\mathtt{Q}$ (specified in LUSTRE and whose truth value is materialized by a boolean signal B), a new program $\mathtt{P}'$ made of the composition of $\mathtt{P}$ and $\mathtt{Q}$ is built, whose only output is B.The LUSTRE compiler produces a graph (an automaton-like code), then verifying the property consists just in verifying that in none of the states the code assigns the value *false* to B. Another method is to compose $\mathtt{P}$ with the equation `assert(Q)` and to verify that $\mathtt{P}$ and $\mathtt{P}'$ have the same inputs and outputs. This approach by constraints is well adapted to SIGNAL: given a program $\mathtt{P} \in \mathcal{P}g_A$ and a property $\mathtt{Q} \in \mathcal{P}g_{A'}$ concerning a subset A' of A, the fact that $\mathtt{Q}$ does not constrain inputs and outputs of $\mathtt{P}$ is expressed by $[\![\mathtt{P}]\!] \parallel [\![\mathtt{Q}]\!] = [\![\mathtt{P}]\!]$. Thanks to Th. 2.1 given on page 14, one obtains as expected that any flow of $\mathtt{P}$ restricted to variables of $\mathtt{Q}$ is a flow of $\mathtt{Q}$:

$$[\![\mathtt{P}]\!] \parallel [\![\mathtt{Q}]\!] = [\![\mathtt{P}]\!] \text{ iff } \textstyle\prod_{A'}([\![\mathtt{P}]\!]) \subseteq [\![\mathtt{Q}]\!] \tag{9}$$

The principle of structural abstraction is similar to the one of synchronous observers: the abstraction of $\mathtt{P}$ is a SIGNAL program $\mathtt{P}_a$ which, composed with $\mathtt{P}$, should not constrain it. An important difference is that the structural abstraction is *inferred from* the program. Moreover a program and its abstraction are defined on the same set of variables: restriction is then useless in (9) which becomes:

$$[\![\mathtt{P}]\!] \parallel [\![\mathtt{P}_a]\!] = [\![\mathtt{P}]\!] \text{ iff } [\![\mathtt{P}]\!] \subseteq [\![\mathtt{P}_a]\!] \tag{10}$$

Formally, a semantically preserving transformation Tr given as a set of rewriting rules turns a SIGNAL program $\mathtt{P} \in \mathcal{P}g_A$ into the parallel composition of two programs $\mathtt{P}_a, \mathtt{P}_r \in \mathcal{P}g_A$:

$$\mathtt{P} \xrightarrow{Tr} \mathtt{P}_a \mid \mathtt{P}_r, \text{ with } [\![\mathtt{P}]\!] = [\![\mathtt{P}_a \mid \mathtt{P}_r]\!] \tag{11}$$

where $\mathtt{P}_a$ aims at representing an abstraction of $\mathtt{P}$, whereas $\mathtt{P}_r$ represents what "remains" as a non-abstracted part of $\mathtt{P}$. The abstraction consists then in keeping $\mathtt{P}_a$ to represent $\mathtt{P}$. The property (10) is verified:

$$[\![\mathtt{P}]\!] \parallel [\![\mathtt{P}_a]\!] = [\![\mathtt{P}]\!] \tag{12}$$

As a matter of fact we have:

$$\begin{array}{rcl} [\![\mathtt{P}]\!] \parallel [\![\mathtt{P}_a]\!] & = & [\![\mathtt{P}_a \mid \mathtt{P}_r]\!] \parallel [\![\mathtt{P}_a]\!] \\ & = & [\![\mathtt{P}_a]\!] \parallel [\![\mathtt{P}_r]\!] \parallel [\![\mathtt{P}_a]\!] \text{ by Def. of } \mid \\ & = & [\![\mathtt{P}_a]\!] \parallel [\![\mathtt{P}_a]\!] \parallel [\![\mathtt{P}_r]\!] \text{ by commutativity of } \parallel \\ & = & [\![\mathtt{P}_a]\!] \parallel [\![\mathtt{P}_r]\!] \text{ by idempotence of } \parallel \\ & = & [\![\mathtt{P}_a \mid \mathtt{P}_r]\!] = [\![\mathtt{P}]\!] \end{array}$$

Moreover the transformation Tr verifies:

- For $\mathtt{P}_1 \in \mathcal{P}g_{A_1}$, $\mathtt{P}_2 \in \mathcal{P}g_{A_2}$, if $Tr(\mathtt{P}_1) = \mathtt{P}_{1a} \mid \mathtt{P}_{1r}$ and $Tr(\mathtt{P}_2) = \mathtt{P}_{2a} \mid \mathtt{P}_{2r}$ then $Tr(\mathtt{P}_1 \mid \mathtt{P}_2) \triangleq \mathtt{P}_a \mid \mathtt{P}_r$ where $\mathtt{P}_a \triangleq \mathtt{P}_{1a} \mid \mathtt{P}_{2a}$ and $\mathtt{P}_r \triangleq \mathtt{P}_{1r} \mid \mathtt{P}_{2r}$;
- Tr is *distributive* over *hiding*: $Tr(\ (|\ \mathtt{P}\ |)/\mathtt{x}\ ) \triangleq (|\ Tr(\mathtt{P})\ |)/\mathtt{x}$;
- Tr is *semantically idempotent* for abstract programs:

$$\text{if } \mathtt{P} \xrightarrow{Tr} \mathtt{P}_a \mid \mathtt{P}_r \text{ and } \mathtt{P}_a \xrightarrow{Tr} \mathtt{P}_{aa} \mid \mathtt{P}_{ar}, \text{ then } [\![\mathtt{P}_{aa}]\!] = [\![\mathtt{P}_a]\!] \tag{13}$$

Let us reformulate structural abstractions in the framework of abstract interpretation by using a Galois insertion.

Theorem 5.1 *Given a (rewriting) transformation $Tr : \mathcal{P}g_A \to \mathcal{P}g_A$, the concrete and abstract domains $\mathsf{Conc} = \mathcal{P}_A$ and $\mathsf{Abs} = \{[\![P_a]\!] \in \mathcal{P}_A \mid \exists P \in \mathcal{P}g_A.\ P \xrightarrow{Tr} P_a \mid P_r\}$, let α_{Tr}, γ_{Tr} be such that:*

$$\begin{array}{rcl} \alpha_{Tr}([\![P]\!]) : \mathsf{Conc} & \to & \mathsf{Abs} \\ [\![P]\!] & \mapsto & [\![P_a]\!] \end{array} \qquad \begin{array}{rcl} \gamma_{Tr} : \mathsf{Abs} & \to & \mathsf{Conc} \\ \varpi & \to & \varpi \end{array}$$

then the pair $(\alpha_{Tr}, \gamma_{Tr})$ is a Galois insertion.

Proof We write α (resp. γ) for α_{Tr} (resp. γ_{Tr}).

1. γ is trivially monotonous. α is monotonous: assume given $\mathtt{P}^1, \mathtt{P}^2 \in \mathcal{P}g_A$ such that $[\![\mathtt{P}^1]\!] \subseteq [\![\mathtt{P}^2]\!]$, $\mathtt{P}^1 \xrightarrow{Tr} \mathtt{P}^1_a \mid \mathtt{P}^1_r$ and $\mathtt{P}^2 \xrightarrow{Tr} \mathtt{P}^2_a \mid \mathtt{P}^2_r$. By Th. 2.1, $[\![\mathtt{P}^1]\!] = [\![\mathtt{P}^1 \mid \mathtt{P}^2]\!]$. Moreover $Tr(\mathtt{P}^1 \mid \mathtt{P}^2) = \mathtt{P}^1_a|\mathtt{P}^2_a \mid \mathtt{P}^1_r|\mathtt{P}^2_r$. Therefore $\alpha([\![\mathtt{P}^1]\!]) = \alpha([\![\mathtt{P}^1 \mid \mathtt{P}^2]\!]) = [\![\mathtt{P}^1_a \mid \mathtt{P}^2_a]\!] = [\![\mathtt{P}^1_a]\!] \parallel [\![\mathtt{P}^2_a]\!] = \alpha([\![\mathtt{P}^1]\!]) \parallel \alpha([\![\mathtt{P}^2]\!])$, and by Th. 2.1 $\alpha([\![\mathtt{P}^1]\!]) \subseteq \alpha([\![\mathtt{P}^2]\!])$.

2. For $\mathtt{P} \in \mathcal{P}g_A$, $[\![\mathtt{P}]\!] \subseteq \gamma \circ \alpha([\![\mathtt{P}]\!])$: $\gamma \circ \alpha([\![\mathtt{P}]\!]) = \alpha([\![\mathtt{P}]\!])$ by definition of γ, and $[\![\mathtt{P}]\!] \subseteq \alpha([\![\mathtt{P}]\!])$ since $[\![\mathtt{P}]\!] \subseteq [\![\mathtt{P}_a]\!]$ by (12).

3. For $\varpi \in \mathsf{Abs}$, $\varpi = \alpha \circ \gamma(\varpi)$: since $\varpi \in \mathsf{Abs}$, $\varpi = \alpha(\varpi')$ for some $\varpi' = [\![\mathtt{P}']\!]$ and $\mathtt{P}' \in \mathcal{P}g_A$. Therefore $\varpi = [\![\mathtt{P}'_a]\!]$. By definition of γ, $\gamma(\varpi) = \varpi$. Since Tr is semantically idempotent for the abstract component (13), $\alpha(\varpi) = [\![\mathtt{P}'_a]\!] = \varpi$. Therefore $\alpha \circ \gamma(\varpi) = \alpha(\varpi) = \varpi$. □

5.1.2 Abstraction by Control

The control part of a program is mainly boolean. The decomposition of a program $\mathtt{P}$ into its control part $\mathtt{P}_\mathcal{C}$ and its data part $\mathtt{P}_\mathcal{D}$ also aims at isolating as much as possible a component composed only of boolean variables, in order to apply well known boolean techniques to it. The decomposition by the transformation $Tr_\mathcal{C}$ is purely syntactical and no new relations is inferred from $\mathtt{P}$: any equation belongs either to $\mathtt{P}_\mathcal{C}$ or to $\mathtt{P}_\mathcal{D}$. We obtain the transformation $Tr_\mathcal{C}$ shown on Fig. 15

$$\mathtt{P} \xrightarrow{Tr_\mathcal{C}} \mathtt{P}_\mathcal{C} \mid \mathtt{P}_\mathcal{D}$$

$Tr_\mathcal{C}$ trivially satisfies conditions (11) and (13).

Boolean variables that are not defined in $\mathtt{P}_\mathcal{C}$ are considered as inputs of the system and are called *free variables*. This decomposition corresponds to a plain disconnection between control and data: no introduction of predicate variables nor computations are performed like in [24] (this point will be discussed in Sect. 5.2.3). We denote by $\mathcal{P}g_\mathcal{C}$ the set of programs $\{\mathtt{P}_\mathcal{C} \mid \mathtt{P} \in \mathcal{P}g\}$. An abstraction by control (also called boolean abstraction) consists then in keeping only the control part of processes:

$$\alpha_{Tr_\mathcal{C}}([\![\mathtt{P}]\!]) = [\![\mathtt{P}_\mathcal{C}]\!]$$

As shown in the following sections, $\mathtt{P}_\mathcal{C}$ and $\mathtt{P}_\mathcal{D}$ can be further decomposed.

P	$\mathtt{P}_{\mathcal{C}}$	$\mathtt{P}_{\mathcal{D}}$
`x := expr` and $D_x \neq \mathbb{B}$	$1_{\mathcal{P}g}$	P
`x := non_bool_expr` and $D_x = \mathbb{B}$	$1_{\mathcal{P}g}$	P
`x := bool_expr` and $D_x = \mathbb{B}$	P	$1_{\mathcal{P}g}$

Fig. 15: Decomposition into Control and Data

5.1.3 Static Abstraction

It was explained in Sect. 2.5.2 that a symbolic transition system establishes a clear distinction between the dynamic part of a process (the transition relation $\rho(\xi, \xi', S)$) and its static part represented by the constraint $\mathcal{C}(S)$. The same decomposition is possible inside a SIGNAL program P, which can be rewritten into the composition of its dynamic part $\mathtt{P}_{\$}$ and its static part $\mathtt{P}_{\mathcal{S}}$. This decomposition is obtained by the transformation $Tr_{\mathcal{S}}$ such that:

$$\mathtt{P} \xrightarrow{Tr_{\mathcal{S}}} \mathtt{P}_{\mathcal{S}} \mid \mathtt{P}_{\$}$$

First note that all dynamic aspects in P are expressed by the delay operator `$`. Therefore if P is a non-delay elementary process (instantaneous function, merge or filtering), then $\mathtt{P}_{\mathcal{S}} = \mathtt{P}$ and $\mathtt{P}_{\$} = 1_{\mathcal{P}g}$. If P is the delay equation `y := x $1 init v0`, the associated constraint $\mathcal{C}(S)$ states that y and x are synchronous (see Eq. (6)), what is expressed in SIGNAL by `y ^= x`. Therefore $\mathtt{P}_{\$} = \mathtt{P}$ and $\mathtt{P}_{\mathcal{S}} =$ `y ^= x`. The *static abstraction* consists then in keeping only the static part of programs:

$$\alpha_{Tr_{\mathcal{S}}}([\![\mathtt{P}]\!]) = [\![\mathtt{P}_{\mathcal{S}}]\!]$$

Since $[\![\mathtt{P}]\!] = [\![\mathtt{P}_{\mathcal{S}} \mid \mathtt{P}_{\$}]\!]$, $Tr_{\mathcal{S}}$ satisfies (11). Moreover $Tr_{\mathcal{S}}$ satisfies (13): it is obvious for non-delay processes since $\mathtt{P}_{\mathcal{S}} = \mathtt{P}$. As far as the delay is concerned, note that the expansion of `y ^= x` into the kernel of SIGNAL given in Sect. 3.1.3 does not involve any delay operator, therefore $[\![\mathtt{y \hat{}= x}]\!] = \alpha_{Tr_{\mathcal{S}}}([\![\mathtt{y \hat{}= x}]\!])$. The static abstraction is summarized on Fig. 16. The fragment of SIGNAL which does not involve the delay operator is

P	$\mathtt{P}_{\mathcal{S}}$
`y := g(x1, ... ,xn)`	`y := g(x1, ... ,xn)`
`y := x $1 init v0`	`y ^= x`
`y := x when c`	`y := x when c`
`y := u default v`	`y := u default v`
$\mathtt{P}^1 \mid \mathtt{P}^2$	${\mathtt{P}^1}_{\mathcal{S}} \mid {\mathtt{P}^2}_{\mathcal{S}}$

Fig. 16: Static abstraction

called *syntactically static*.

Definition 5.1 (Syntactically static* SIGNAL*) *The fragment of the kernel of* SIGNAL *composed of the parallel composition | and of the elementary processes: instantaneous function, filtering and deterministic merge is called (the)* syntactically static SIGNAL *(fragment), written* $\mathcal{S}$SIGNAL.

The abstraction of $\mathtt{P} \in \mathcal{P}g_A$ induced by $Tr_{\mathcal{S}}$ exactly gives a static process in the sense of Sect. 2.4.3. If we denote by ϖ the process $\alpha_{Tr_{\mathcal{S}}}([\![\mathtt{P}]\!])$, then $\varpi = \varpi_{\mathcal{S}}$: indeed the flow semantics of non-delay elementary processes expresses in function of a single instant t, with no reference to instants that precede or follow it. Moreover in the case of the delay: if P denotes `y := x $1 init v0`, $\mathtt{P}_a$ denotes `y ^= x` and D is the domain of x and y, then

$$\begin{aligned}[\![\varpi_{\mathtt{P}}]\!]\mathcal{V}_A &= \{(\star,\star),(v_0,v_x),(v_y,v_x) \mid v_y, v_x \in D\} \\ &= \{(\star,\star),(v_y,v_x) \mid v_y, v_x \in D\} \\ &= [\![\varpi_{\mathtt{P}_a}]\!]\mathcal{V}_A\end{aligned}$$

Note that a static program does not necessarily belong to the syntactically static fragment. Just consider the process `mx := x $ 1 | x := mx` which is static in the sense of Def. 2.22. The following mainly deals with static programs specified in $\mathcal{S}$SIGNAL. When no confusion can occur, a static program refers to a syntactically static one.

5.1.4 Abstraction by Synchronizations

The abstraction by synchronizations applies mainly to the data part $\mathtt{P}_{\mathcal{D}}$ of programs, since as shown in Sect. 5.2 the static control part can be expressed by synchronizations without loss of semantics. Any SIGNAL equation specifies a relation between variables, that is a relation between their clocks (how their instants of presence are related with each others) and their values (what relations hold between values at instants of presence). Relations between clocks induced by a program `P`, called *synchronizations* of `P`, can be described by a SIGNAL program denoted by $\mathtt{P}_{\mathcal{C}k}$, as follows. The clock of a variable x is obtained by the expression `event x` (see Sect. 3.1.3). The synchronization between two variables y and x is expressed by `event y ^= event x` or `y ^= x`. Assume to simplify that the considered processes do not involve any constant (or they are represented by a variable as explained in Rem. 3.2, which allows to give them a clock). Synchronizations are shown on Fig. 17. We denote

P	$\mathtt{P}_{\mathcal{C}k}$
`y := g(x1, ... ,xn)`	`(event y ^= event x1) | ... | (event y ^= event xn)`
`y := x $1 init v0`	`event y ^= event x`
`y := x when c`	`event y ^= ( (event x) when c )`
`y := u default v`	`event y ^= ( (event u) default (event v) )`

Fig. 17: Synchronizations

by $\mathcal{P}g_{\mathcal{C}k}$ the set of programs $\{\mathtt{P}_{\mathcal{C}k} \mid \mathtt{P} \in \mathcal{P}g\}$. The transformation $Tr_{\mathcal{C}k}$ is such that

$$\mathtt{P} \xrightarrow{Tr_{\mathcal{C}k}} \mathtt{P}_{\mathcal{C}k} \mid \mathtt{P}$$

This decomposition highlights the richness of SIGNAL: it is easy to describe relations between clocks in a program. The handling of clocks is direct, in accordance with their importance in data-flow languages, and is all the more flexible as numerous derived operators [23] are provided to simplify the specification of clock constraints (e.g. exclusiveness, inclusion, etc).

The *abstraction by synchronizations* consists then in only keeping relations between clocks (that correspond to largest possible relations):

$$\alpha_{Tr_{\mathcal{C}k}}([\![\mathtt{P}]\!]) = [\![\mathtt{P}_{\mathcal{C}k}]\!]$$

This abstraction looses all information about the values carried by variables in case of presence. Note that it is itself an abstraction of the static abstraction presented in Sect. 5.1.3: $[\![\mathtt{P}_{\mathcal{S}}]\!] \subseteq [\![\mathtt{P}_{\mathcal{C}k}]\!]$.

Let us verify that the transformation satisfies conditions (11) and (13). $Tr_{\mathcal{C}k}$ is semantically preserving. Let us show on the simple case of the delay that $Tr_{\mathcal{C}k}$ satisfies (13): if `P` is `event y ^= event x` then the abstraction of $\varpi = [\![\mathtt{P}]\!]$ is equivalent to ϖ. As a matter of fact `P` rewrites into `(| b1 := when y=y | b2 := when x=x | b := b1 = b2 |)/ b,b1,b2`, then $\mathtt{P}_{\mathcal{C}k}$ is

```
(| event b1 ^= event y | event b2 ^= event x
 | event b ^= event b1 | event b ^= event b2 |)/ b,b1,b2
```

By applying the transitivity of `^=` and by forgetting variables `b`, `b1` and `b2`, one checks that $[\![\mathtt{P}_{\mathcal{C}k}]\!] = [\![$ `event y ^= event x` $]\!]$.

5.1.5 Composition of Abstractions

Abstractions presented in sections 5.1.2, 5.1.3 and 5.1.4 are not used as such in POLYCHRONY but composed to provide finer ones which we generally call *clock abstractions*. Two main cases are presented here. Both take into account the control and synchronization parts of programs. In the first case (Fig. 18(a)), the dynamic part is kept to perform model-checking and controller synthesis in the tool SIGALI [38]. A separation between control

($\mathtt{P}_{\mathcal{C}}$) and data ($\mathtt{P}_{\mathcal{D}}$) is first performed, then synchronizations are extracted from the data part. The abstraction consists then in only keeping $\mathtt{P}_{\mathcal{C}} \mid \mathtt{P}_{\mathcal{DC}k}$ on P, where $\mathtt{P}_{\mathcal{DC}k}$ means $(\mathtt{P}_{\mathcal{D}})_{\mathcal{C}k}$. In the second case (Fig. 18(b)) only the static part is kept for a boolean analysis of admissible valuations. The static part is extracted from the control and data part, then synchronizations are extracted from the static data part. The abstraction consists then in keeping only $\mathtt{P}_{\mathcal{CS}} \mid \mathtt{P}_{\mathcal{DSC}k}$ on P. Sect. 5.2 explains how it can be converted into a purely boolean system.

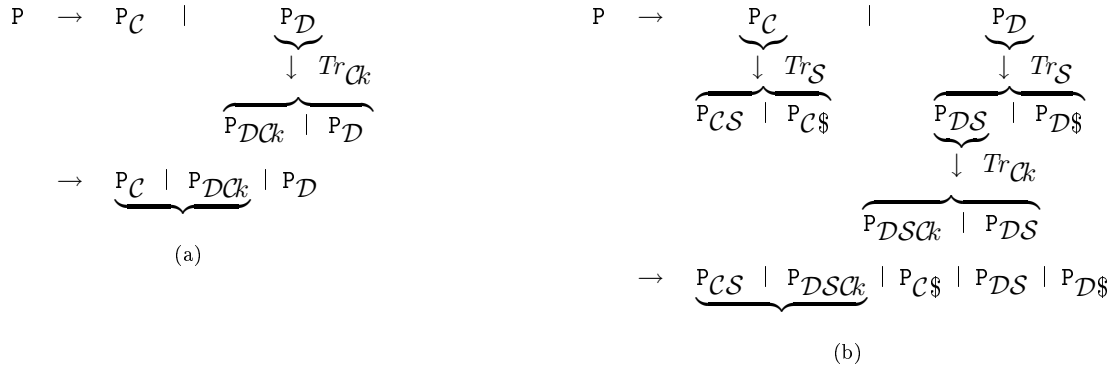

Fig. 18: Composition of Structural Abstractions in POLYCHRONY

More generally, the idea is to obtain a chain of abstractions and verification tools whose complexity is increasing: abstractions are applied from the coarsest to the finest until the minimal degree of complexity to prove the property is found. For example, a static abstraction can be refined into a dynamic one; the only consideration of synchronizations can be refined into the consideration of the full control part, etc. In the framework of structural abstractions, it is also possible to over-step the boolean abstraction by e.g. decomposing the data part $\mathtt{P}_{\mathcal{D}}$ into a decidable part $\mathtt{P}_{\mathcal{D}_d}$ (e.g. all instantaneous functions fall into the Presburger algebra) and a non-decidable one $\mathtt{P}_{\mathcal{D}_{nd}}$ (e.g. containing non linear expressions). Finally, semantical abstractions can be applied on data domains with an increasing complexity: intervals, regions, polyhedra etc.

5.2 Boolean Structural Abstractions

We show in this section that abstractions by control and by synchronizations presented in Sect. 5.1 are boolean: abstract processes can be encoded into a purely boolean transition system.

In the classical sense a specification is *boolean* if it involves only *boolean variables* (that is variables of the type $\mathbb{B} = \{true, false\}$) and *boolean operators* such as *not*, *or*, *and*. The boolean abstraction presented in Sect. 4.2.1 produces boolean specifications with this meaning. More generally, a boolean reasoning involves variables whose data type is not necessarily $\mathbb{B}$, but contains only two values.

The description of a data-flow specification that contains only boolean variables (e.g. programs in $\mathcal{P}g_{\mathcal{C}}$) semantically necessitates three values, since all domains are extended with absence. A good illustration is the encoding of a process into a three-valued logic performed by [38]. Nevertheless, any three-valued logic can be encoded into a boolean algebra by *artificially* increasing the number of variables, as shown in Ex. 5.1.

Example 5.1 Assume given a domain $D = \{v_1, v_2, v_3\}$ for variables in S. The value $v \in D$ of a variable $x \in S$ can be encoded into the domain $D' = \{v_1, v_1'\}$ with an auxiliary *don't care* variable x', which plays no role in the specification other than indicating values taken by x. The value of x' does not matter if $x = v_1$, but indicates whether x is v_2 or v_3 when x is assigned v_1':

$$x = v_1, x' =? \rightsquigarrow x = v_1 \qquad x = v_1', \begin{cases} x' = v_1 \rightsquigarrow x = v_2 \\ x' = v_1' \rightsquigarrow x = v_3 \end{cases}$$

◇

On the other hand, the status of a variable x, expressed in the data type of *pure events* $\{ true, \star \}$ or more generally in $\{present, absent\}$, is basically boolean. Therefore synchronizations and in particular programs in $\mathcal{P}g_{\mathcal{C}k}$ can also be expressed into a boolean algebra. Dedicated to the description of static processes, the clock

algebra associated to a valuation (see Sect. 2.4.3) encodes naturally synchronizations as well as the static part of boolean programs. We explain in Sect. 5.2.1 how it can be encoded into the propositional calculus and in Sect. 5.2.2 how programs in $\mathcal{P}g_{\mathcal{C}\mathcal{C}k} = \mathcal{P}g_{\mathcal{C}} \cup \mathcal{P}g_{\mathcal{C}k}$ are described. Finally the link between this boolean abstraction and the predicate abstraction presented in Sect. 4.2.1 in discussed in Sect. 5.2.3.

5.2.1 Clock Algebra and Propositional Calculus

As mentioned in Sect. 2.4.3, the language $\langle U, \cap, \cup, \setminus \rangle$ that describes clocks can be interpreted with respect to a valuation into a boolean algebra with two elements (the empty set and a singleton). Such an algebra is sufficient to describe static boolean processes. The propositional calculus is the most natural framework, and is used in POLYCHRONY. Out of habit and by abuse of terms, "the" Clock Algebra (with capital letters) refers to the language composed of variables in $\mathcal{K}$, the symbol $\mathbb{0}$ and operators $\cup$, $\cap$, $\setminus$ and $\subseteq$, interpreted into the propositional calculus.

$\mathcal{K}$ is a finite set of clocks called *clock variables*. The encoding of a program into the Clock Algebra is a system of *clock equations* over $\mathcal{K}$, called *clock system*. A clock system is encoded into the propositional calculus as follows: any clock variable $\widehat{x}$ is encoded into the propositional variable $b_{\widehat{x}}$. The semantics of the clock system relates a valuation V to a distribution δ_V over $b_{\mathcal{K}}$: $\delta_V(b_{\widehat{x}}) = 1$ (resp. 0) means "x is present (resp. absent) in V". The relation $\widehat{x} \subseteq \widehat{y}$ is then encoded into $b_{\widehat{x}} \Rightarrow b_{\widehat{y}}$. By extension and using the natural semantics of characteristic functions, $\widehat{x} \cup \widehat{y}$ is encoded into $b_{\widehat{x}} \vee b_{\widehat{y}}$, $\widehat{x} \cap \widehat{y}$ into $b_{\widehat{x}} \wedge b_{\widehat{y}}$, and $\widehat{x} \setminus \widehat{y}$ into $b_{\widehat{x}} \wedge \neg b_{\widehat{y}}$. The empty clock is encoded by the constant that has no model, that is *false*. A more formal presentation is given in Sect. 6.4.

5.2.2 Encoding into the Clock Algebra

Let us first explain how the Clock Algebra encodes synchronizations. We use a mapping $\mathcal{CA}$ which encodes directly SIGNAL programs `P` into a clock system, in which a comma is used to separate equations. The encoding is direct in the case of synchronizations inferred from the elementary processes instantaneous function, delay and deterministic merge (cf Sect. 3.1): one just needs to replace in Fig. 17 expressions of the kind `event y` (which delivers the clock of y) by $\widehat{y}$, see Fig. 19(a). On the other hand, the filtering involves a predicate (the

P	$\mathcal{CA}$(P)
`y := g(x1, ... ,xn)`	$\widehat{y} = \widehat{x_1}, \ldots, \widehat{y} = \widehat{x_1}$
`y := x $1 init v0`	$\widehat{y} = \widehat{x}$
`y := u default v`	$\widehat{y} = \widehat{u} \cup \widehat{v}$

(a)

P	$\mathcal{CA}$(P)
`y := x when c`	$\widehat{y} = \widehat{x} \cap [c]$, $[c] \cup [\neg c] = \widehat{c}$, $[c] \cap [\neg c] = \mathbb{0}$
$P_1 \mid P_2$	$\mathcal{CA}(P_1), \mathcal{CA}(P_2)$

(b)

Fig. 19: Encoding of synchronizations into the Clock Algebra

guard), represented by a boolean variable in the kernel of SIGNAL, say $c \in S$, that must be encoded into a clock variable.

At a given instant c can be absent or present: it is indicated by the value of the abstract variable $\widehat{c}$. When present, c takes the value *true* or *false*. An auxiliary clock variable written $[c] \in \mathcal{K}$ is then associated to c: it denotes the set of instants where c is *present and true*. The set of instants where c is present and false is then denoted by another auxiliary clock variable written $[\neg c]$. Such clock variables in square brackets are called *clocks-condition*. Clocks $[c]$ and $[\neg c]$ are mutually constrained since obviously in terms of set of instants they form a partition of $\widehat{c}$. This partition is specified in the encoding of the filtering into the Clock Algebra shown on Fig. 19(b).

Remark 5.1 If we represent the domain $\{1, 0\}$ by $\{$*present, absent*$\}$, we have the following scheme:

$$b_{\widehat{c}} = b_{[c]} = b_{[\neg c]} = \textit{absent} \rightsquigarrow c = \star \qquad b_{\widehat{c}} = \textit{present}, \begin{cases} b_{[c]} = \textit{absent},\ b_{[\neg c]} = \textit{present} \rightsquigarrow c = \textit{false} \\ b_{[c]} = \textit{present},\ b_{[\neg c]} = \textit{absent} \rightsquigarrow c = \textit{true} \end{cases}$$

It is an encoding of the three values $\{$*true*, *false*, $\star\}$ into the two ones $\{$*present, absent*$\}$ by means of auxiliary variables (condition-clocks). Compared to the one of Ex. 5.1, the encoding based on the Clock Algebra is very

natural: it fully respects the data-flow principle which is to first wondering if a variable is present and next considering its value. Note that the values of $b_{[c]}$ and $b_{[\neg c]}$ are fixed and have a meaning when c is absent: they are absent as well. Note also that actually only one auxiliary variable is needed: for example $[\neg c] = \widehat{c} \setminus [c]$.

Example 5.2 We give the clock system associated to the absolute value (Ex. 3.2). Let us define the boolean variables c_p and c_n, that encode the predicates $c_p = y \geq 0$ and $c_n = y < 0$. We have:

$$\begin{array}{lll} \widehat{p} = [c_p] \cap \widehat{y} & [c_p] \cap [\neg c_p] = \mathbb{O} & [c_p] \cup [\neg c_p] = \widehat{y} \\ \widehat{n} = [c_n] \cap \widehat{y} & [c_n] \cap [\neg c_n] = \mathbb{O} & [c_n] \cup [\neg c_n] = \widehat{y} \\ \widehat{a} = \widehat{p} \cup \widehat{n} & & \end{array}$$

◇

Let us now explain how synchronizations encode boolean relations between variables, e.g. specified by programs in $\mathcal{P}g_{\mathcal{C}}$. Filtering and merge processes are encoded by clock systems as follows, for boolean variables y, u, v, c, x:

$$\texttt{y := x when c} \xrightarrow{\mathcal{CA}} \left\{ \begin{array}{l} \widehat{y} = \widehat{x} \cap [c] \\ [y] = [x] \cap [c] \end{array} \right. \quad \text{and } \texttt{y := u default v} \xrightarrow{\mathcal{CA}} \left\{ \begin{array}{l} \widehat{y} = \widehat{u} \cup \widehat{v} \\ [y] = [u] \cup ([v] \setminus \widehat{u}) \end{array} \right.$$

Then convenient SIGNAL rewritings transform instantaneous boolean relations (built on operators *and*, *or*, *not*) into synchronizations composed of expressions `event c` and operators `default`, `when` and `^=`. For example `y := a and b` rewrites into `y := a when a when b default not event a default not event b`. These synchronizations are themselves encoded into the Clock Algebra using condition-clocks.

A dynamic system can be represented by a SLTS (see Sect. 2.5.2) whose constraint $\mathcal{C}$ is encoded into the propositional calculus. For a delay equation `y := x $ 1 init v0` we obtain using the SLTS $\Theta = (b_{\mathcal{K}}, b_\xi, \theta, \rho, \mathcal{C})$ where $b_{\mathcal{K}} = \{b_{\widehat{y}}, b_{\widehat{x}}, b_{[x]}, b_{[\neg x]}, b_{[y]}, b_{[\neg y]}\}$ and $b_\xi = \{\xi_{b_{[x]}}\}$:

$$\theta(b_\xi) : (\xi_{b_{[x]}} = v_0) \quad , \quad \mathcal{C}(b_{\mathcal{K}}) : \left(\begin{array}{l} b_{\widehat{y}} = b_{\widehat{x}} \\ b_{[x]} \vee b_{[\neg x]} = b_{\widehat{x}} \\ \mathit{not}\ (b_{[x]} \wedge b_{[\neg x]}) \\ b_{[y]} \vee b_{[\neg y]} = b_{\widehat{y}} \\ \mathit{not}\ (b_{[y]} \wedge b_{[\neg y]}) \end{array} \right) \quad , \quad \rho(b_\xi, b_{\mathcal{K}}) : \left\{ \begin{array}{l} \xi'_{b_{[x]}} = \left\{ \begin{array}{l} b_{[x]} \textbf{ if } b_{\widehat{x}} \\ \xi_{b_{[x]}} \textbf{ else} \end{array} \right. \\ \textbf{if } b_{\widehat{y}} \textbf{ then } b_{[y]} = \xi_{b_{[x]}} \end{array} \right.$$

The following focuses only on static aspects.

5.2.3 Link with Predicate Abstraction

Guards predicates that appear in transition systems analyzed by [24, 52] have a direct counterpart in SIGNAL: it is the *guards of filtering*. For example, the equation `p := y when y>=0` corresponds intuitively to the conditional affectation **if** *y is present* **then if** $y \geq 0$ **then** $p := y$, where $y \geq 0$ plays indeed the role of a guard. Composed abstractions presented in Sect. 5.1.5 are then a particular case of the predicate abstraction presented in Sect. 4.2, in the sense that the abstract specification contains only boolean variables, some of which correspond to predicates of the concrete specification. However the choice of abstract predicates is not free but restricted to guards of filtering, thus determined by the specification. Therefore the abstraction is far less precise than it could be.

As a matter of fact, in classical predicate abstraction any control point of the specification is examined, and it is checked whether variables satisfy the relations (or their negation) abstracted by predicates at this point. On the contrary, clock abstractions of non boolean equations considers only control points that correspond to filtering. No formal work exists which applies the predicate abstraction to SIGNAL programs. Nevertheless, let us admit it can be done and consider an intuitive example.

Example 5.3 Let us consider the system `(| y := x+1 when x>10 | z := 4 when y > 5 |)`. Predicates $x > 10$ and $y > 5$ are distinguished and represented respectively by $B_{x>10}$ and $B_{y>5}$. The first equation expresses that **if** $x > 10$ **then** $y := x + 1$. The abstraction function presented in Sect. 4.2.1 detects that just after the affectation $y := x+1$ the predicate $B_{y>5}$ is satisfied, since the formula $(x > 10 \wedge y = x+1) \Rightarrow y > 5$ is a tautology. On the contrary, the abstraction by synchronizations leads to the only clock equation $\widehat{y} = \widehat{x} \cap [B_{x>10}]$. A flow F such that for all $t \in \mathbb{N}$: $z_t^F = \star$, $y_t^F < 5$ and $x_t^F > 10$ is then a model of the abstract program. Without loss of precision, one expects to obtain the following equations

$$\widehat{y} = \widehat{x} \cap [B_{x>10}] \ , \ \widehat{y} = [B_{y>5}]$$

that indicate that y is always greater than 5. ◇

Clock abstractions are therefore coarser than predicate abstraction, nevertheless they still give a good approximation of the control of the system. Indeed, the clock of a variable can be seen as a guard for its values: **if** *x is present* **then** *compute on* v_x. By extension, synchronizations contribute to represent the control of the program: the abstraction by control is indeed refined by an abstraction by synchronizations in POLYCHRONY.

5.3 Analyses of Admissible Valuations

Recall that as explained while presenting SLTS in Sect. 2.5.2, the static part of a process ϖ is characterized by the set of its *admissible valuations*, that are solutions of the constraint $\mathcal{C}(S)$ in SLTS. It is a super-set of $[\![\varpi]\!]_{\mathcal{V}_A}$, the set of valuations that can effectively occur along a trajectory. The analysis of trajectories is itself fundamental, but the analysis of admissible valuations is also possible, though its usefulness is often misjudged. We first give two reasons that motivate such an analysis, then briefly present a practical analysis commonly used in POLYCHRONY.

Verification of Properties As explained in introduction, it is sometimes already possible to verify some instantaneous properties on the static part of a process, before the consideration of its trajectories. In the general case the analysis of the static part is not sufficient, but it is worth trying it, since it has a complexity far lower than the additional consideration of trajectories. It is all the more interesting since properties invariant over the time are the most important ones in reactive systems.

Practically, the analysis of admissible valuations allows to verify "static properties", meaning that do not depend on the dynamic of the system, e.g. in Ex. 3.2 that a is always positive. As shown in Ex.8.3, it is sometimes also possible to check some simple *inductive* invariant properties, i.e. safety properties true in the initial state of the system and preserved by all transitions.

Role of $\mathcal{C}(S)$ in Analyses The second argument is proper to the synchronous paradigm and illustrated here by the data-flow formalism. As already said before, computations theoretically involve two implicit stages: one must first determine which variables are present (the *status* of variables), and only then use these values in computations. As an example, let us consider the classical execution of a step forward in a SLTS, given a particular valuation for memories. One must first look for a valuation V_S that satisfies $\mathcal{C}(S)$ and such that *present* delay variables are valued by memories (see Eq. (6): **if** $x \neq \star$ **then** $y = \xi_x$). Then memories that correspond to a *present* signal variable are updated (see Eq. (6): **if** $x \neq \star$ **then** $\xi'_x = x$ **else** $\xi'_x = \xi_x$). In general, these two-stages computations force to adopt a special representation for the constraint $\mathcal{C}(S)$ that indicates separately the status and the values of variables. The computation of this representation corresponds to a preliminary resolution of $\mathcal{C}(S)$, that is to an analysis of the admissible valuations of the system.

In practice, the need for such a representation and resolution depends on the way absence in handled. It is illustrated here by the analyses of [38] and [11] presented in Sect. 3.2.1 and 3.2.2, which differentiate themselves by this handling, among other things.

[38] distinguishes three possible values for signal variables in a boolean SLTS: absent, present and *true*, present and *false*. It encodes these possibilities into the three-valued logic $\mathbb{Z}/3\mathbb{Z}$, in such a way that each value of the logic contains in itself all information needed to use it: a status (*absent* for 0, *present* for 1 and -1) and a value (no value for 0, *true* for 1 and *false* for -1). The encoding of absence and presence is therefore homogeneous. The transition relation $\rho(\xi', \xi, S)$, the constraint $\mathcal{C}(S)$ and the initialization predicate are then encoded into polynomials over the field $\mathbb{Z}/3\mathbb{Z}$. The encoding of a SIGNAL equation is direct, and parallel composition yields to easy aggregation of polynomials. All computations are then encoded into computations on polynomials handling uniformly absence and presence: there is no need of the above mentioned decomposition into two stages.

On the contrary [11] cannot use such a uniform representation: it is not possible to encode absence into a particular value of $\mathbb{N}$ (or $\mathbb{Q}$). The use of a representation that indicates the status of variables, and then the value of present variables is imperative. [11] has chosen to encode the status into lists of *absent* variables, combined with polyhedra that encode relations between *present* variables. The computation of such an hybrid representation is called *normalization*. It is trivial for a single equation, but complex as soon as parallel composition is involved, contrary to the case of polynomials. After the normalization, the two stages needed to perform a step forward are: 1. Intersection of a polyhedron which represents a set of states with a polyhedron

which represents a set of valuations restricted to present variables; 2. Projection of the result on memories that correspond to a present variable.

So in the general case, any analysis of the trajectories of a system necessitates an analysis of its admissible valuations. Moreover, as shown below, such an analysis — thought not strictly necessary for [38] — allows to increase significantly its performances. Generally speaking, any analysis can take advantage of a preliminary analysis of admissible valuations.

Clock Calculus The *clock calculus* [2] is historically the first analysis designed in POLYCHRONY, implemented in the SIGNAL compiler. It is an analysis of the admissible valuations of a program P, applied to its abstraction $\mathtt{P}_{\mathcal{CS}}$ | $\mathtt{P}_{\mathcal{DSCK}}$ presented in Sect. 5.1.5. This abstraction is described using the Clock Algebra and encoded into a boolean system as explained in Sect. 5.2.1. Just as it uses a structural abstraction, the clock calculus performs a *structural analysis*: it realizes a set of semantically preserving transformations of the clock system (or the constraint $\mathcal{C}(\mathcal{K})$), then modifying its structure. Just as [11] modifies the structure of $\mathcal{C}(S)$ to distinguish status of variables from relations between present variables, the clock calculus gives a "good" shape to the system. "Good" means that after transformation it is easy to verify some properties and to perform given operations, namely to

- determine if the process is *endochronous* (see Sect. 2.4.2), that is deterministic and executable in an asynchronous environment (the analysis then contains a *decision procedure*);
- verify the equivalence or *inclusion* of some *clocks*, when needed during the decision procedure (it can then perform a kind of *model-checking*);
- *generate code* (the decision procedure is constructive, since it synthesizes the control of the program from which the code generation is direct).

Although the clock calculus applies to a plain system of boolean equations, it does *not* have as its ambition to be a tool that, given a boolean property, checks if it is true or not of the system. The compiler is a *formal proof system* which is both more than such a boolean tool (decision procedure for testing endochrony, generation of code) and less (only some boolean properties exhibited by the new shape of the system are checkable, which represent more or less the set of properties needed to test endochrony for the average of programs). Efficiency is stressed and heuristics are used: the compilation time should remain acceptable.

The clock calculus involves many complex choices for data structures and algorithms which are not detailed here: we just note that BDD are used. Experimentations show that the calculus induces a judicious order on boolean variables of BDD that significantly reduces their size. Just consider the measures showed in Fig. 20 that can be found in [2]. For each sample SIGNAL program the number of boolean variables is given, as well as the total sum of nodes involved in BDD for: in first row the representation computed by the clock calculus (trees), in second row the representation by the characteristic function of $\mathcal{C}(\mathcal{K})$ for some order of variables, in third row the representation by the characteristic function of $\mathcal{C}(\mathcal{K})$ for the order determined by the calculus. The tool SIGALI is directly connected to the compiler: before the analysis of trajectories, the analysis on admissible valuations is performed by the compiler and the order between variables is among others transmitted to SIGALI. As for BDD this order reduces significantly the size of TDD used to encode the transition relation. Model-checking and controller synthesis are then all the more efficient. The analysis of static specifications is thus proved to prepare efficiently the analysis of trajectories.

5.4 Abstractions into Data Domains

Structural abstractions are attractive by their simplicity: the abstract program is obtained either by picking some equations according to simple criteria (e.g. decomposition into control and data, isolation of equations that involve only linear expressions) or by very simple transformations that do not necessitate semantical computations (static abstraction, abstraction by synchronizations). More complex abstractions are needed, in particular that approximate the behavior of processes using abstract data domains such as monomials on boolean variables, real intervals or polyhedra.

As said in Sect. 2.5 and 3.2, a process can be represented by a symbolic transition system close to a classical one whose domain is extended with the special value $\star$ to denote absence of variables. Numerous analyses dedicated to transition systems exist: one should only need to adapt them so that their consider extended domains to obtain analyses for data-flow programs. Following this idea [11] adapts the work of [21] on the abstract domain of polyhedra. Similar analyses could be designed for other domains mentioned in Sect. 4.2 (intervals, regions, Presburger formulas, etc). In all cases clocks should not appear as objects of analyses.

	Trees	Char. function for some order	Char. function for the tree order
Stopwatch 1318 var	61893	unable-cpu	unable-cpu
Watch 785 var	34753	unable-cpu	unable-cpu
Alarm 465 var	3428	unable-mem	unable-cpu
Chrono 282 var	1548	unable-mem	422975
Supervisor 202 var	425	unable-cpu	226472
Pace Maker 96 var	50	53610	582
Robot 99 var	36	unable-cpu	415

Fig. 20: Nodes of BDD depending on analyses of $\mathcal{C}(\mathcal{K})$

But the analysis of [11] shows that the adaptation is not so easy: the normalization of the constraint $\mathcal{C}(S)$ was necessary. The problem is even encountered at the semantical level: let us illustrate it by considering the simple abstract data domain of intervals $(\mathcal{I}, \sqsubseteq_{\mathcal{I}}, \sqcup_{\mathcal{I}}, \sqcap_{\mathcal{I}})$. The principle is to associate to a set of variables $A \subseteq S$ a set of abstract variables A_I such that $|A| = |A_I|$ and $x^I \in A_I$ is an variable of type $\mathcal{I}^\star$ whose value in case of presence surrounds the possible values carried by x. The standard Galois insertion for intervals ignoring notion of absence given by Cousot in [20] states that, for a set of values $X = \{v_j \in \mathbb{N} \mid j \in J\}$

$$\alpha : X \ \rightarrow \ [minX, maxX] \ \ , \ \ \gamma : [m, M] \ \rightarrow \ \{v \in \mathbb{N} \mid m \leq v \leq M\}$$

Let us try to apply the same Galois insertion to our models that are valuations: $\mathsf{Conc} = \mathcal{P}(\mathcal{V}_A)$ and $\mathsf{Abs} = \mathcal{P}(\mathcal{V}_{A^I})$. Concretization is easy: given $V^I \in \mathcal{V}_{A^I}$,

$$\gamma(\{V^I\}) = \{V \in \mathcal{V}_A \mid \text{ for all } x \in A, \textbf{ if } V^I(x^I) = \star \textbf{ then } V(x) = \star \textbf{ else } V(x) \in V^I(x^I)\}$$

The handling of absence makes things more complicated to abstract a set $\mathcal{V}^c \in \mathcal{P}(\mathcal{V}_A)$. For a given variable $x \in A$: if x is absent in any $V \in \mathcal{V}^c$ then x^I is absent in any $V^I \in \alpha(\mathcal{V}^c)$; if x is present in any $V \in \mathcal{V}^c$ then x^I is present and surrounds values taken by x in $\mathcal{V}^c$. If x is absent and present in $\mathcal{V}^c$ then both previous cases should be considered: one should abstract separately sets of valuations in which x is present resp. absent as follows

$$\textbf{if } V(x) = \star \text{ for all } V \in \mathcal{V}^c \textbf{ then } V^I(x^I) = \star$$

$$\textbf{if } V(x) \neq \star \text{ for all } V \in \mathcal{V}^c \textbf{ then } V^I(x^I) = [min\ \{V(x)\}_{V \in \mathcal{V}^c}\ ,\ max\ \{V(x)\}_{V \in \mathcal{V}^c}]$$

More generally $\mathcal{V}^c$ should be split into partitions according to the status of variables: we recognize here again the normalization of [11].

Because the problem of considering both status and values of variables is recurrent (since characteristic of synchronous languages) and because the status of variables is naturally represented by clocks, we propose a new approach for non-structural abstractions. Its principle is to introduce clocks as objects of analyses and use them to handle synchronizations, while taking into consideration values of non-boolean variables, thus over-stepping the expressiveness of the Clock Algebra. Only static aspects are addressed in a first approach: the extension to dynamic aspects should use sLTS. Notions informally presented here are an introduction to and motivate the formalism of the clock language given in Sect. 6.

Let us consider for example the Presburger algebra as an abstract domain and see how data-flow processes can be described using synchronizations and Presburger formulas. We denote by $g^P : \mathbb{Z} \times \cdots \times \mathbb{Z} \rightarrow \mathbb{Z}$ a safe abstraction of $g : D \times \cdots \times D \rightarrow D$. In a first attempt, let us intuitively approximate elementary processes as shown on Fig. 21. The abstract specification is composed on the one hand of a *clock system* as the one described in Sect. 5.2.2 (which indicates the relative status of variables) and on the other hand of a Presburger formula (which solutions indicate what values can be carried by present variables). This solution is simple but clearly not satisfactory, as shown by the following example.

$$
\begin{array}{rcl}
\texttt{y := g(x}_1\texttt{, ... ,x}_n\texttt{)} & \rightarrow & \widehat{y} = \widehat{x_1} = \cdots = \widehat{x_n},\ \ y = g^P(x_1, \ldots, x_n) \\
\texttt{y := x \$ 1 init v0} & \rightarrow & \widehat{y} = \widehat{x} \\
\texttt{y := x when c} & \rightarrow & \begin{cases} \widehat{y} = \widehat{x} \cap [c] \\ [c] \cup [\neg c] = \widehat{c} \\ [c] \cap [\neg c] = \mathbb{O} \end{cases}, \ \ y = x \\
\texttt{y := u default v} & \rightarrow & \widehat{y} = \widehat{u} \cup \widehat{v},\ \ y = u \vee y = v
\end{array}
$$

Fig. 21: Synchronizations plus Presburger formulas

Example 5.4 Let us consider the program `(| y := u default v | u := x+1 | v := u-2 |)`. The obtained abstract clock system is $\widehat{y} = \widehat{u} \cup \widehat{v}, \widehat{u} = \widehat{x}, \widehat{v} = \widehat{u}$ and the Presburger formula relating values of variables is $(y = u \vee y = v) \wedge u = x + 1 \wedge v = u - 2$. They indicate that $\widehat{y} = \widehat{x}$ and that when present their values satisfy $y = x + 1 \vee y = x - 1$. But the concrete program only authorizes y to take the value $x + 1$, since v is never present when u is absent. ◇

To express a deterministic merge one must specify that y takes the value of u *when* u *is present*, else the value of v. There is a causal relation between the presence of u and the satisfaction of $y = u$: it is the classical interaction between control and values carried by variables. We then need to *mix* clocks and Presburger formulas.

Let us use the isomorphism between the Clock Algebra and the propositional calculus presented in Sect. 5.2.1. If we assume given a theory that embeds the propositional calculus and the Presburger algebra, we should abstract elementary processes by *hybrid* formulas of this theory as shown on Fig. 22.

$$
\begin{array}{rcl}
\texttt{y := g(x}_1\texttt{, ... ,x}_n\texttt{)} & \rightarrow & b_{\widehat{y}} \Leftrightarrow b_{\widehat{x_1}} \Leftrightarrow \ldots \Leftrightarrow b_{\widehat{x_n}} \ \wedge \ (\ b_{\widehat{y}} \Rightarrow (y = g^P(x_1, \ldots, x_n)) \ ) \\
\texttt{y := x \$ 1 init v0} & \rightarrow & b_{\widehat{y}} \Leftrightarrow b_{\widehat{x}} \\
\texttt{y := x when c} & \rightarrow & \begin{cases} b_{\widehat{y}} \Leftrightarrow (b_{\widehat{x}} \wedge b_{[c]}) \\ (b_{[c]} \wedge b_{[\neg c]}) \Leftrightarrow b_{\widehat{c}} \ \wedge (\ b_{\widehat{y}} \Rightarrow (y = x) \ ) \\ \neg(b_{[c]} \wedge b_{[\neg c]}) \end{cases} \\
\texttt{y := u default v} & \rightarrow & (\ (b_{\widehat{y}} \Leftrightarrow (b_{\widehat{u}} \vee b_{\widehat{v}}) \ ) \ \wedge \ (\ b_{\widehat{u}} \Rightarrow (y = u) \ ) \ \wedge \ (\ (b_{\widehat{v}} \wedge \neg b_{\widehat{u}}) \Rightarrow (y = v) \ )
\end{array}
$$

Fig. 22: Synchronizations mixed to Presburger formulas

Hybrid formulas are intuitive but quite far from the flow semantics: it is not easy to understand whether all of them have a meaning, alternatively if they can be handled as standard formulas, as shown by Ex. 5.5.

Example 5.5 Assume given the formulas on $\{x, y, z\}$ ϕ_1: $(\ b_{\widehat{z}} \Rightarrow (x \geq y) \ ) \wedge (\ b_{\widehat{z}} \Rightarrow (x < y) \ )$ and ϕ_2: $(\ (x \geq y) \Rightarrow b_{\widehat{z}} \ ) \wedge (\ (x < y) \Rightarrow b_{\widehat{z}} \ )$ that state a constraint on the values of x and y, and the status of z. We have:

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\phi_1 & \Leftrightarrow & b_{\widehat{z}} \Rightarrow (x \geq y \wedge x < y) & \phi_2 & \Leftrightarrow & (x \geq y \vee x < y) \Rightarrow b_{\widehat{z}} \\
& \Leftrightarrow & b_{\widehat{z}} \Rightarrow \mathit{false} & & \Leftrightarrow & \mathit{true} \Rightarrow b_{\widehat{z}} \\
& \Leftrightarrow & \neg b_{\widehat{z}} & & \Leftrightarrow & b_{\widehat{z}}
\end{array}
$$

The result is expected for ϕ_1: $b_{\widehat{z}}$ is constrained to be *false*, what corresponds to $\widehat{z} = \mathbb{O}$ (in terms of the Clock Algebra). Intuitively, if z is present then x and y can only carry contradictory values, therefore z is always absent. On the contrary, the result is surprising for ϕ_2: this time $b_{\widehat{z}}$ is constrained to be *true*, what means that z is always present ! In our mind ϕ_2 just states that *when* x *and* y *are present* with a non-constrained value, then z is present. It can be expressed by the formula $(b_{\widehat{x}} \wedge b_{\widehat{y}}) \Rightarrow b_{\widehat{z}}$. ◇

If one wishes that the handling of hybrid formulas like ϕ_2 in Ex. 5.5 gives results in accordance with intuition, one should remember that Presburger formulas actually denote relations between *present* variables: e.g. $x \geq y \vee x < y$ denotes the clock $\widehat{x} \cap \widehat{y}$ and should be rewritten into $b_{\widehat{x}} \wedge b_{\widehat{y}}$. In this case the interaction between clocks and values of variables must be explicitly managed and mixed formulas cannot be handled by standard methods. Alternatively, one could consider only mixed formulas in which relations between values of variables are syntactically binded to clocks: e.g. for Ex. 5.5 $(\ ((b_{\widehat{x}} \wedge b_{\widehat{y}}) \Rightarrow (x \geq y)) \Rightarrow b_{\widehat{z}} \ ) \wedge (\ ((b_{\widehat{x}} \wedge b_{\widehat{y}}) \Rightarrow (x < y)) \Rightarrow b_{\widehat{z}} \ )$ is indeed equivalent to $(b_{\widehat{x}} \wedge b_{\widehat{y}}) \Rightarrow b_{\widehat{z}}$.

These alternatives are two aspects of the same problem: any relation between values of variables implicitly assumes that variables are present: e.g. **if** x *and* y *are present* **then** $x < y$. There already exists a formalism that makes this assumption semantically explicit: the condition-clock $[c]$ denotes instants where c *is present* and its *value* is *true*. A natural formalization of hybrid formulas is an extension of the Clock Algebra by new clock terms called relation-clocks. Denoted by $\langle R(x_1, \ldots, x_n)\rangle$, they embed a general predicate R on variables $\{x_1, \ldots, x_n\}$ and denote valuations such that variables x_i are present and their values satisfy R. The remainder of this document is fully dedicated to the formal definition of a clock language which provides such relation-clocks.

6 A Clock Language

Like the Clock Algebra presented in Sect. 5.2.1 the clock language describes instantaneous properties of processes by clock inclusions in a flow. Nevertheless it is far more expressive since it extends the Clock Algebra with clock terms which embed predicates of a first order language and describe relations between variables. Notations are given in Sect. 6.1, then the $\mathcal{CL}$ language is presented in Sect. 6.2 and 6.3, and a boolean abstraction of $\mathcal{CL}$ useful to design its decision procedure is discussed in Sect. 6.4.

6.1 Notations

Notations presented in the previous sections remain valid. We focus here on notations needed to describe general predicates that involve symbols of constant, variable and predicate. We assume given a first order language $\mathcal{L}$ with equality (see e.g. [18, 37]) whose set of symbols of variable contains S. The set of formulas of $\mathcal{L}$ is denoted by $\mathcal{F}(\mathcal{L})$, whose typical elements are R, R_1, R_2 (e.g. R is $x_1 < x_2 + 10$, where $x_1, x_2 \in S$). The set of free variables of $R \in \mathcal{F}(\mathcal{L})$ is denoted by $fv(R)$ and we note $R(x_1, \ldots, x_n)$ to emphasize that $fv(R) = \{x_1, \ldots, x_n\}$. A structure (or a realization) $\mathcal{M}$ for $\mathcal{L}$ is obtained by choosing a domain D and an interpretation for symbols of predicate and function according to D (e.g. D is $\mathbb{Z}$, $+$ is the addition over integers). Given an interpretation function $l : S \to D$ for variables in S, we write $\mathcal{M}, l \models\!\!\!\approx R$ or simply $l \models\!\!\!\approx R$ (when $\mathcal{M}$ is clear) if R is satisfied by l, or $(d_1, \ldots, d_n) \models\!\!\!\approx R(x_1, \ldots, x_n)$ for $d_i \in D, i = 1 \ldots n$. In the following we assume chosen a fixed structure for $\mathcal{L}$.

All that concerns the $\mathcal{CL}$ language is clearly reminiscent of the Clock Algebra. In particular terms of the $\mathcal{CL}$ language (*clock terms*) presented in Sect. 6.2 denote clocks, while *clock formulas* presented in Sect. 6.3 denote inclusions between clock terms.

6.2 Clock Terms

Like in the Clock Algebra a clock term can be a clock variable, the empty clock, or a composition of terms by set-like operators. The novelty resides in terms called *relation-clocks* that embed predicates of $\mathcal{F}(\mathcal{L})$. Then a clock term h is described as follows[10]:

$$h ::= \mathbb{O} \mid \widehat{x} \mid \langle R(x_1, \ldots, x_n)\rangle \mid h_1 \cap h_2 \mid h_1 \cup h_2 \mid h_1 \setminus h_2$$

where $R \in \mathcal{F}(\mathcal{L})$. The empty clock $\mathbb{O}$, clock variables $\widehat{x}$ and relation-clocks $\langle R\rangle$ are *basic* clock terms, whose set is denoted by $\mathcal{BCT}$, or $\mathcal{BCT}_A$ when variables can only belong to A. An *inductive* clock term has the form h_1 *op* h_2 where *op* $\in \{\cup, \cap, \setminus\}$. The set of clock terms on A is denoted by $\mathcal{CT}_A$, whose typical elements are h, h_1, h_2, h^1, etc. The set of all clock terms is denoted by $\mathcal{CT}$.

Let us now interpret clock terms as snapshots of execution. Whereas the semantics of Clock Algebra was informally given in Sect. 5.2.1 in terms of distributions since it is isomorphic to the propositional calculus, clock terms involve values of variables and their semantics is given directly in terms of valuations.

It was shown in Sect. 2.2.4 that clocks are directly linked to valuations inside flows by means of their characteristic function: for $x \in A$ a valuation $V \in \mathcal{V}_A$ denotes an instant of the clock of x if $V(x) \neq \star$, and then V is said to *switch on* the clock of x. A valuation $V \in \mathcal{V}_A$ therefore induces an unary predicate $on_V(\cdot)$ over $\mathcal{CT}_A$: $on_V(h)$ is true iff V switches on h, for $h \in \mathcal{CT}_A$. Valuations described by clock terms are then defined by means of this predicate, by induction over the structure of h:

Definition 6.1 *Given $A \subseteq S$ and $V \in \mathcal{V}_A$, $on_V(\cdot)$ is defined for basic clock terms by:*

- *not* $on_V(\mathbb{O})$;

[10] We use BNF in which the symbol | denotes choice in the grammar and has nothing to do with the syntactical operator of parallel composition |.

- *for* $x \in A$, $on_V(\widehat{x})$ *iff* $V(x) \neq \star$;
- *for* $\{x_1, \ldots, x_n\} \subseteq A$, $on_V(\ \langle R(x_1, \ldots, x_n)\rangle\ )$ *iff for* $i = 1 \ldots n$, $on_V(\widehat{x_i})$ *and* $(V(x_1), \ldots, V(x_n)) \approx R$;

Given $A_1, A_2 \subseteq A$, $h_1 \in \mathcal{CT}_{A_1}$, $h_2 \in \mathcal{CT}_{A_2}$, *and* $V \in \mathcal{V}_A$, $on_V(h_1\ op\ h_2)$ *is defined by:*

- $on_V(h_1 \cap h_2)$ *iff* $on_V(h_1)$ *and* $on_V(h_2)$;
- $on_V(h_1 \cup h_2)$ *iff* $on_V(h_1)$ *or* $on_V(h_2)$;
- $on_V(h_1 \setminus h_2)$ *iff* $on_V(h_1)$ *and not* $on_V(h_2)$.

Example 6.1 Assume given $A = \{x, y\}$ and let V be defined by $V(x) = 3$ and $V(y) = \star$. $on_V(\widehat{y})$ is false, $on_V(\widehat{x} \setminus \widehat{y})$ is true, $\langle x > 0\rangle$ is switched on in V but $\langle x > 0 \vee y > 0\rangle$ is switched off, since $\widehat{y}$ is switched off (y is absent in V). $\diamond$

6.3 Clock Formulas

Clock formulas describe predicates over clock terms, namely inclusions. Like for processes, the $\mathcal{CL}$ language supplies composition (corresponding to conjunction) and restriction (corresponding to existential quantification), but also negation. A clock formula ϕ is then described by:

$$\phi ::= h_1 \subseteq h_2 \ \mid\ \phi_1 \wedge \phi_2 \ \mid\ \exists x.\phi \ \mid\ \neg\phi$$

where $h_1, h_2 \in \mathcal{CT}$ and $x \in S$. For $A \subseteq S$, the set of clock formulas on A is denoted by $\mathcal{CF}_A$, whose typical elements are ϕ, ϕ_1, ϕ^1, etc. The set of all clock formulas is denoted by $\mathcal{CF}$. The *clock term universe* associated to $\phi \in \mathcal{CF}_A$ and denoted by $CT(\phi) \subseteq \mathcal{CT}_A$ is defined inductively: $CT(h_1 \subseteq h_2)$ is the set of clock sub-terms that compose h_1 and h_2; $CT(\exists x.\phi) = CT(\neg\phi) = CT(\phi)$ and $CT(\phi_1 \wedge \phi_2) = CT(\phi_1) \cup CT(\phi_2)$. We denote by $BCT(\phi)$ the set $\mathcal{BCT} \cap CT(\phi)$ and by $CF(\phi) \subseteq \mathcal{CF}$ the set of sub-formulas of ϕ. The set of quantifier-free formulas (that contain no sub-formulas of the kind $\exists x.\phi$) in $\mathcal{CF}$ (resp. $\mathcal{CF}_A$) is denoted by $\mathcal{CF}^{qf}$ (resp. $\mathcal{CF}^{qf}{}_A$). In the following, we simply write $h_1 \equiv h_2$ for $h_1 \subseteq h_2 \wedge h_2 \subseteq h_1$[11].

The semantics of clock formulas is as announced based on valuations.

Definition 6.2 *Given* $A \subseteq S$, $V \in \mathcal{V}_A$, $A_1, A_2 \subseteq A$, $h_1 \in \mathcal{CT}_{A_1}$, $h_2 \in \mathcal{CT}_{A_2}$, $\phi, \phi_1 \in \mathcal{CF}_{A_1}$ *and* $\phi_2 \in \mathcal{CF}_{A_2}$:

- $V \models h_1 \subseteq h_2$ *iff* $on_V(h_1)$ *implies* $on_V(h_2)$;
- $V \models \phi_1 \wedge \phi_2$ *iff* $V \models \phi_1$ *and* $V \models \phi_2$;
- $V \models \neg\phi$ *iff* $V \not\models \phi$.

Given $x \in S$, $A_3 \subseteq S$ *s.t.* $A_3 \setminus \{x\} \subseteq A$, *and* $\phi \in \mathcal{CF}_{A_3}$:

- $V \models \exists x.\phi$ *iff there exists* $V' \in \mathcal{V}_{A_3}$ *s.t.* $V' \models \phi$ *and* $V'_{|A_3 \setminus \{x\}} = V_{|A_3 \setminus \{x\}}$.

A formula $\phi \in \mathcal{CF}_A$ *is* valid, *written* $\models \phi$, *whenever* $V \models \phi$ *for all* $V \in \mathcal{V}_A$.

As expected the interpretation of clock formulas can be lifted to flows (resp. to processes) as follows: given a formula $\phi \in \mathcal{CF}_A$, we say that a flow $F \in \mathcal{F}_A$ (resp. a process $\varpi \in \mathcal{P}_A$) satisfies ϕ, written again $F \models \phi$ (resp. $\varpi \models \phi$) whenever ϕ holds for all valuations $V \in [\![F]\!]_{\mathcal{V}_A}$ (resp. $V \in [\![\varpi]\!]_{\mathcal{V}_A}$).

To clarify notations we define a function $\Upsilon(\cdot)$ (read "clock of") which associates a clock term to a formula $R \in \mathcal{F}(\mathcal{L})$:

$$\Upsilon(R) \text{ is the clock term } \bigcap\{\widehat{x} \mid x \in fv(R)\}.$$

Intuitively, "the clock of R" is the greatest set of instants (for the set inclusion) where all variables whose value is required to strictly evaluate R are present. Note that $\Upsilon(\neg R)$ is equal to $\Upsilon(R)$ (since formulas R and $\neg R$ have same free variables) and that by definition $on_V(\Upsilon(R))$ iff $on_V(\widehat{x})$, for all $x \in fv(R)$. By abuse of notations we write $V \approx R$ when all variables in $fv(V)$ are present in V and carry values that satisfy R.

Example 6.2 $\Upsilon(x < 10) = \Upsilon(\ \exists z.(z \geq 0 \wedge x < 10 + z)\ ) = \widehat{x}$ and in Ex 6.1 it is true that $V \approx x < 10$, just as $V \approx \exists z.(z \geq 0 \wedge x < 10 + z)$. On the contrary, $\Upsilon(x > 0 \vee y > 0) = \widehat{x} \cap \widehat{y}$ and it is false that $V \approx x > 0 \vee y > 0$. $\diamond$

[11] The new notation $\equiv$ is introduced to denote equivalence of clocks, since $=$ is kept to denote equality of the first order language $\mathcal{L}$.

The semantics of relation-clocks can be rewritten more concisely using Υ:

$$on_V(\langle R\rangle) \text{ iff } on_V(\Upsilon(R)) \text{ and } V \approx\!\!\models R \tag{14}$$

Therefore, for any valuation V, $on_V(\langle R\rangle)$ *implies* $on_V(\Upsilon(R))$, namely $\models \langle R\rangle \subseteq \Upsilon(R)$; also it can be proved that $\models \Upsilon(R) \equiv \langle R\rangle \cup \langle \neg R\rangle$, as well as $\models \langle R_1\rangle \cap \langle R_2\rangle \equiv \langle R_1 \wedge R_2\rangle$, etc. Such tautologies are very useful to compute on clocks, and could address axiomatization issues, which is out of the scope of this document.

Example 6.3 The $\mathcal{CL}$ language can express various fine instantaneous properties. Assume given $x, y \in S$.

1. "x is always positive" is expressed by the formula $\widehat{x} \equiv \langle x \geq 0\rangle$, or equivalently $\widehat{x} \subseteq \langle x \geq 0\rangle$, since $\Upsilon(x \geq 0) = \widehat{x}$ hence $\models \langle x \geq 0\rangle \subseteq \widehat{x}$.
2. "x is always even" is expressed by $\widehat{x} \subseteq \langle \exists y.x = 2y\rangle$, and highlights the need for quantification provided by $\mathcal{L}$.
3. "y is always equal to x" can be expressed by $\widehat{y} \equiv \langle y = x\rangle$ or equivalently $\widehat{y} \subseteq \langle y = x\rangle$: y can be absent while x is present.
4. (a) "y is always equal to x when x is positive" can be expressed by $\langle x \geq 0\rangle \subseteq \langle y = x\rangle$, meaning "if x is positive then y is present and equal to x, if x is absent then y can be absent or present with any value, and similarly if x is present and negative". In this last case y can even be equal to x, e.g. $V(x) = V(y) = -2$.
   (b) On the contrary $\langle x \geq 0\rangle \equiv \langle y = x\rangle$ expresses "x is positive iff y is equal to x": y cannot be equal to x when x is negative.
   (c) Finally $\langle x \geq 0\rangle \equiv \langle y = x\rangle \equiv \widehat{y}$ expresses "y is present iff x is positive, and y is always equal to x", which is the expected behavior of a statement like `if x ≥ 0 then y := x` in the case where it completely defines `y`.

◇

Remark 6.1 The clock language $\mathcal{CL}$ extends the Clock Algebra thanks to relation-clocks, that are a generalization of condition-clocks $[c]$ used to describe the boolean abstraction of programs in Sect. 5.2.2. If we abuse notations the clock term $[R(x_1, \ldots, x_n)]$ implicitly constrains variables $x_1, \ldots, x_n$ to be synchronous (i.e. $\Upsilon(R) \equiv \widehat{x_1} \equiv \ldots \equiv \widehat{x_n}$), since the symbol R is interpreted as a SIGNAL instantaneous function (or relation) as defined in Sect. 3.1.1: such functions only allow the construction of monochronous predicates. It is not the case of relation-clocks $\langle R\rangle$ (e.g. $\langle x + y > 0\rangle$ lets free the clocks of x and y) since R is this time interpreted as a symbol of $\mathcal{L}$, thus ignoring notion of synchronization. ◇

Remark 6.2 The $\mathcal{CL}$ language provides the negation of a clock formula (useful in Sect. 8.2), as a meta negation when $\mathcal{CF}$ is interpreted over valuations but not when interpretation is lifted to flows: for $\phi \in \mathcal{CF}$, $F \models \neg\phi$ iff $\forall V \in [\![F]\!]_{\mathcal{V}_S}, V \models \neg\phi$, which differs from $F \not\models \phi$ ($\exists V \in [\![F]\!]_{\mathcal{V}_S}.V \not\models \phi$). There is no counterpart in the Clock Algebra. ◇

6.4 Boolean Abstraction

Unlike the Clock Algebra, the $\mathcal{CL}$ language also involves values of non-boolean variables, therefore it can only be *abstracted* into the propositional calculus. Such a boolean abstraction is presented here. It uses notations informally introduced while encoding the Clock Algebra into the propositional calculus in Sect. 5.2.2, then can be seen as a clean formalization of this encoding if relation-clocks are forgotten.

Let us first give a few notations for the propositional calculus $\mathbf{B}$. Given $\mathbf{b}$ a set of propositional variables, $\delta : \mathbf{b} \to \{0, 1\}$ is a *distribution* of values on $\mathbf{b}$ which naturally extends to formulas of the full propositional calculus in the standard way (e.g. $\delta(G_1 \wedge G_2) = 1$ iff $\delta(G_1) = 1$ and $\delta(G_2) = 1$). Δ denotes the set of distributions on $\mathbf{b}$, and $\mathbf{0}$ denotes the distribution mapping all variables to 0. For $\delta \in \Delta$ and $G \in \mathbf{B}$, we write $\delta \models G$ whenever $\delta(G) = 1$.

$\mathcal{CL}$ can be *abstracted* into $\mathbf{B}$, where $\mathbf{b}$ contains a variable $b_{\widehat{x}}$ (resp. $b_{\langle R\rangle}$) for each $\widehat{x}$ (resp. $\langle R\rangle$) in $\mathcal{BCT}$. The abstraction is a mapping $B : \mathcal{CT} \cup \mathcal{CF} \to \mathbf{B}$ which maps $h \in \mathcal{CT}$ (resp. $\phi \in \mathcal{CF}$) to h_B (resp. ϕ_B) instead of $B(h)$ (resp. $B(\phi)$) as follows:

Definition 6.3 (Boolean Abstraction) *The mapping B is defined by induction on the structure of elements in $\mathcal{CT}$ and $\mathcal{CF}$ on Fig. 23, where $x \in S$ and $\exists b_{\langle R_x \rangle}.\phi_B$ is written for $\exists b_{\langle R_1 \rangle} \ldots b_{\langle R_n \rangle}\phi_B$ in which $\{R_i\}_{i=1..n} = \{R \in \mathcal{F}(\mathcal{L}) \mid \langle R \rangle \in BCT(\phi)$ and $x \in fv(R)\}$.*

$h \in \mathcal{CT}$	$h_B \in \mathbf{B}$
$\widehat{x}$	$b_{\widehat{x}}$
$\langle R \rangle$	$b_{\langle R \rangle}$
$\mathbb{O}$	*false*
$h^1 \cap h^2$	$h^1_B \wedge h^2_B$
$h^1 \cup h^2$	$h^1_B \vee h^2_B$
$h^1 \setminus h^2$	$h^1_B \wedge \neg h^2_B$

$\phi \in \mathcal{CF}$	$\phi_B \in \mathbf{B}$
$h^1 \subseteq h^2$	$h^1_B \Rightarrow h^2_B$
$\neg\phi$	$\neg\ \phi_B$
$\exists x.\phi$	$\exists b_{\widehat{x}}.b_{\langle R_x \rangle}.\phi_B$
$\phi^1 \wedge \phi^2$	$\phi^1_B \wedge \phi^2_B$

Fig. 23: Abstraction of $\mathcal{CL}$

For $h \in \mathcal{CT}$, we say that δ switches on h whenever $\delta(h_B) = 1$.

We now study properties of models of concrete and abstract *quantifier-free* formulas. We consider the concrete domain of sets of valuations ($\mathsf{Conc} = \mathcal{P}(\mathcal{V}_A)$) and the abstract domain of sets of distributions ($\mathsf{Abs} = \mathcal{P}(\Delta)$). We first define the *abstraction* of a valuation V as the distribution $\delta_V \in \Delta$ which indicates whether variables are required to be present or absent in V, and which formulas $R \in \mathcal{F}(\mathcal{L})$ are required to be satisfied by the present ones, for $\langle R \rangle \in BCT(\phi)$.

Definition 6.4 (Abstraction) *Let $V \in \mathcal{V}_A$. $\delta_V \in \Delta$ is defined by:*

$$\textit{for all } h \in \mathcal{CT}_A,\ \delta_V(h_B) = 1 \textit{ iff } on_V(h).$$

Remark 6.3 It is easy to prove that Def. 6.4 delivers indeed a distribution, e.g. for $h^1, h^2 \in \mathcal{CT}_A$, $\delta_V(h^1_B \wedge h^2_B) = 1$ iff $\delta_V(h^1_B) = 1$ and $\delta_V(h^2_B) = 1$. Indeed $\delta_V(h^1_B \wedge h^2_B) = 1$ iff $\delta_V((h^1 \cap h^2)_B) = 1$ by definition of B, iff $on_V(h^1 \cap h^2)$, iff $on_V(h^1)$ and $on_V(h^2)$ by definition of $on_V(\cdot)$, iff $\delta_V(h^1_B) = 1$ and $\delta_V(h^2_B) = 1$.

Theorem 6.1 states that a quantifier-free clock formula ϕ is satisfied by a valuation V iff its abstraction is satisfied by δ_V, the abstraction of V.

Theorem 6.1 *For all $\phi \in \mathcal{CF}^{qf}{}_A$ and $V \in \mathcal{V}_A$, $V \models \phi$ iff $\delta_V \models \phi_B$.*

Proof The proof is by induction on the structure of ϕ.
Basic Case For $h, h' \in \mathcal{CT}_A$, assume $V \models h \subseteq h'$. Equivalently "$on_V(h)$ iff $on_V(h')$" by Def. 6.2. Equivalently we have "$\delta_V(h_B) = 1$ iff $\delta_V(h'_B) = 1$" by Def. 6.4, iff $\delta_V(h_B \Rightarrow h'_B) = 1$, iff $\delta_V((h \subseteq h')_B) = 1$ by Def. 6.3, iff $\delta_V \models (h \subseteq h')_B$.
Induction Step
Negation: for $\phi \in \mathcal{CF}^{qf}$, assume $V \models \neg\phi$, equivalently *not* $V \models \phi$ by Def. 6.2, and *not* $\delta_V \models \phi_B$ by induction hypothesis. Equivalently $\delta_V(\neg\phi_B) = 1$, iff $\delta_V((\neg\phi)_B) = 1$ by Def. 6.2, iff $\delta_V \models (\neg\phi)_B$.
Composition: Given $A_1, A_2 \subseteq A$, $\phi^1 \in \mathcal{CF}^{qf}{}_{A_1}$ and $\phi^2 \in \mathcal{CF}^{qf}{}_{A_2}$, assume $V \models \phi^1 \wedge \phi^2$, equivalently $V \models \phi^1$ *and* $V \models \phi^2$, and $\delta_V \models \phi^1_B$ *and* $\delta_V \models \phi^2_B$ by induction hypothesis. Equivalently $\delta_V \models \phi^1_B \wedge \phi^2_B$, iff $\delta_V \models (\phi^1 \wedge \phi^2)_B$. □

Reciprocally, we define the *concretization* of a distribution $\delta \in \Delta$ as the set of valuations $\mathcal{V}_\delta$. It contains all valuations that assign to variables in A values in $D^\star$, *when possible*, in accordance with: on the one hand the requirements of presence resp. absence of variables specified by δ; on the other hand the satisfaction of some formulas in $\mathcal{F}(\mathcal{L})$ imposed by the values of variables $b_{\langle R \rangle}$. Formally,

Definition 6.5 (Concretization) *Given $\delta \in \Delta$, $\mathcal{V}_\delta \subseteq \mathcal{V}_A$ is defined by*

$$\mathcal{V}_\delta = \{V \in \mathcal{V}_A \mid \forall h \in \mathcal{BCT}_A, on_V(h) \textit{ iff } \delta(h_B) = 1\}.$$

Like for Def 6.4, the definition of $\mathcal{V}_\delta$ generalizes to clock terms in $\mathcal{CT}_A$. Note that $\mathcal{V}_\delta$ can be empty for some $\delta \in \Delta$, what means that δ cannot be made concrete. It is the case e.g. if $\mathbf{b}$ is $\{b_{\widehat{x}}, b_{\langle x<0 \rangle}, b_{\langle x>3 \rangle}\}$ and $\delta(b_{\widehat{x}}) = \delta(b_{\langle x<0 \rangle}) = \delta(b_{\langle x>3 \rangle}) = 1$. As a matter of fact, some $V \in \mathcal{V}_\delta$ must be s.t. $V(x) = v$ for some $v \in D_x$ (since $\delta(b_{\widehat{x}}) = 1$) with $v < 0$ (since $\delta(b_{\langle x<0 \rangle}) = 1$) and $v > 3$ (since $\delta(b_{\langle x>3 \rangle}) = 1$).

As one can expect, these abstraction and concretization of models define a Galois insertion, useful in Sect. 7.1.

Theorem 6.2 *Let α and γ be the following mappings:*

$$\alpha: \begin{array}{ccc} \mathcal{P}(\mathcal{V}_A) & \rightarrow & \mathcal{P}(\Delta) \\ \{V_i\}_{i\in I} & \mapsto & \{\delta_{V_i}\}_{i\in I} \end{array} \quad \textit{and} \quad \gamma: \begin{array}{ccc} \mathcal{P}(\Delta) & \rightarrow & \mathcal{P}(\mathcal{V}_A) \\ \Delta' & \mapsto & \bigcup_{\delta\in\Delta'} \mathcal{V}_\delta \end{array}$$

then the pair (α, γ) is a Galois insertion.

Proof α and γ are trivially monotonous.

• $Id_{\mathsf{Conc}} \subseteq \gamma \circ \alpha$: let $\{V_i\}_{i\in I}$ be a set of valuations. Then $\alpha(\{V_i\}_{i\in I}) = \{\delta_{V_i}\}_{i\in I}$ by definition of α. By Def. 6.4, for each V_i and each $h \in \mathcal{BCT}$, $on_{V_i}(h)$ iff $\delta_{V_i}(h_B) = 1$. Therefore, by Def 6.5, $V_i \in \mathcal{V}_{\delta_{V_i}}$. As a consequence, $\{V_i\}_{i\in I} \subseteq \bigcup_{i\in I} \mathcal{V}_{\delta_{V_i}}$. Since $\bigcup_{i\in I} \mathcal{V}_{\delta_{V_i}} = \gamma(\{\delta_{V_i}\}_{i\in I}) = \gamma\circ\alpha(\{V_i\}_{i\in I})$ by definition of γ, $\{V_i\}_{i\in I} \subseteq \gamma\circ\alpha(\{V_i\}_{i\in I})$.

• $\alpha \circ \gamma = Id_{\mathsf{Abs}}$: let Δ' be a set of distributions. $\gamma(\Delta') = \bigcup_{\delta\in\Delta'} \mathcal{V}_\delta$ by definition of γ. By Def. 6.4 and 6.5, for all V in some $\mathcal{V}_\delta$, we have $\delta_V = \delta$. Therefore $\alpha(\mathcal{V}_\delta) = \{\delta\}$, so $\alpha(\bigcup_{\delta\in\Delta'} \mathcal{V}_\delta) = \Delta' = \alpha \circ \gamma(\Delta')$. □

7 A Decision Procedure

In this section we define a decision procedure for the satisfiability of a clock formula $\phi \in \mathcal{CF}$. It answers YES if ϕ is satisfiable: there exists a non-trivial flow F (that contains at least a non-silent valuation) such that $F \models \phi$; and NO otherwise. Since satisfaction for flows relies on satisfaction for all valuations of a flow, there exists a flow which satisfies ϕ iff there exists a valuation which satisfies ϕ. The procedure proceeds as follows: the boolean abstraction ϕ_B of ϕ is computed and leads to a finite number of decision problems in the theory $\mathcal{F}(\mathcal{L})$. It is checked for each model of ϕ_B whether its concretization is not empty, hence contains a model of ϕ. Section 7.1 gives technical materials for the decision procedure which is discussed in Sect. 7.2.

For technical reasons we consider particular *completed* formulas:

Definition 7.1 (Completed formula) *The* completion of *the formula $\phi \in \mathcal{CF}$ is the* completed *formula* $\phi \wedge \bigwedge\{\Upsilon(R) \equiv \langle R\rangle \cup \langle\neg R\rangle \mid \langle R\rangle \in BCT(\phi)\}$

Note that the additional sub-formulas are a partial counterpart of the braced equations given on Fig. 19(b) for condition-clocks. Restricting to completed formulas is no loss of generality since any formula and its completion have the same models.

Theorem 7.1 *For all $V \in \mathcal{V}_S, V \models \phi$ iff $V \models \phi \wedge \bigwedge\{\Upsilon(R) \equiv \langle R\rangle \cup \langle\neg R\rangle \mid \langle R\rangle \in BCT(\phi)\}$.*

Proof Let us prove that for all $V \in \mathcal{V}_S$ and $\langle R\rangle \in BCT(\phi)$, $V \models \Upsilon(R) \equiv \langle R\rangle \cup \langle\neg R\rangle$. By Def. 6.1 $on_V(\langle R\rangle \cup \langle\neg R\rangle)$ iff $on_V(\langle R\rangle)$ *or* $on_V(\langle\neg R\rangle)$. Since $\Upsilon(R)$ is equal to $\Upsilon(\neg R)$ and by Eq. (14), $on_V(\langle R\rangle \cup \langle\neg R\rangle)$ is then equivalent to $on_V(\Upsilon(R))$ *and* $(V \not\approx R$ *or* $V \not\approx \neg R)$, therefore to $on_V(\Upsilon(R))$ by the excluded middle. Finally $on_V(\langle R\rangle \cup \langle\neg R\rangle)$ is equivalent to $on_V(\Upsilon(R))$ and $V \models \Upsilon(R) \equiv \langle R\rangle \cup \langle\neg R\rangle$. □

7.1 The Satisfiability Problem

We now study the satisfiability problem for clock formulas.

Corollary 7.1 *For all $\phi \in \mathcal{CF}^{qf}{}_A$ and for all $V \in \mathcal{V}_A$, $V \models \phi$ iff $V \in \bigcup_{\delta\models\phi_B} \mathcal{V}_\delta$.*

Proof $\Rightarrow$) Assume given $V \in \mathcal{V}_A$ s.t. $V \models \phi$. By Th 6.1, $\delta_V \models \phi_B$ and by Th. 6.2, $V \in \gamma \circ \alpha(\{V\}) = \gamma(\{\delta_V\}) = \mathcal{V}_{\delta_V}$. Therefore $V \in \bigcup_{\delta\models\phi_B} \mathcal{V}_\delta$.
$\Leftarrow$) Assume given $\delta \in \Delta$ and $V \in \mathcal{V}_\delta$. Then $\delta_V \in \{\delta_{V'} \mid V' \in \mathcal{V}_\delta\} = \alpha(\mathcal{V}_\delta)$. Since $\gamma(\{\delta\}) = \mathcal{V}_\delta$, $\delta_V \in \alpha \circ \gamma(\{\delta\})$. Then by Th. 6.2 $\delta = \delta_V$. If moreover $\delta \models \phi_B$, we obtain by Th. 6.1 $V \models \phi$. □

Corollary 7.1 states that if some $V \models \phi$, then $V \in \mathcal{V}_\delta$ for some $\delta \models \phi_B$, and vice versa. For each $\delta \models \phi_B$, we establish a criterion for the non-emptiness of $\mathcal{V}_\delta$, which relies on the satisfiability of a first order formula R_δ built up from ϕ and δ (see Th. 7.2). In the rest of this section, we assume given a completed formula $\phi \in \mathcal{CF}^{qf}{}_A$.

We define $\mathcal{R}_\delta$ as the set of formulas $R \in \mathcal{F}(\mathcal{L})$ such that $\langle R\rangle \in BCT(\phi)$ is switched on in δ. Such formulas must be satisfied by values of variables assigned by any valuation which is a concretization of δ.

Definition 7.2 *Let $\mathcal{R}_\delta$ be the set $\{R \mid \langle R\rangle \in BCT(\phi)$ and $\delta(b_{\langle R\rangle}) = 1\}$. When $\mathcal{R}_\delta$ is not empty, R_δ denotes the conjunction of all formulas in $\mathcal{R}_\delta$.*

The following lemma ensures that distributions we consider are well-formed, that is the values of $b_{\widehat{x}}$ for $x \in fv(R)$ are coherent with the values of $b_{\langle R\rangle}$.

Lemma 7.1 *Let $\delta \models \phi_B$. For all $R \in \mathcal{R}_\delta$ and for all $x \in fv(R)$, $\delta(b_{\widehat{x}}) \neq 0$.*

Proof Since ϕ is completed, for each $\langle R\rangle \in BCT(\phi)$ the sub-formula $\Upsilon(R) \equiv \langle R\rangle \cup \langle \neg R\rangle$, say ψ in the following, occurs positively in ϕ. Hence $\delta \models \phi_B$ implies $\delta \models \psi_B$, where ψ_B is

$$\underbrace{(\bigwedge_{x \in fv(R)} b_{\widehat{x}})}_{\Upsilon(R)_B} \Leftrightarrow b_{\langle R\rangle} \vee b_{\langle \neg R\rangle}. \tag{15}$$

If $R \in \mathcal{R}_\delta$ then $\delta(b_{\langle R\rangle}) = 1$ and by Eq. 15 we conclude. □

Theorem 7.2 gives a criterion for the non-emptiness of $\mathcal{V}_\delta$. Finally its corollary (Cor. 7.2) states that the satisfiability of ϕ is fully determined by the pairs (δ, R_δ), where $\delta \models \phi_B$.

Theorem 7.2 *Let $\delta \models \phi_B$. Then, $\mathcal{V}_\delta \neq \emptyset$ iff "$\mathcal{R}_\delta = \emptyset$ or else R_δ is satisfiable". Moreover $\forall V \in \mathcal{V}_\delta$, $V \approx\!\!\models R_\delta$.*

Proof When $\mathcal{R}_\delta$ is an empty set the theorem simply states that $\mathcal{V}_\delta$ is not empty. Indeed, one can choose an arbitrary value $v \in D$ and define W by

$$W(x) = \begin{cases} v & \text{if } \delta(b_{\widehat{x}}) = 1 \\ \star & \text{otherwise} \end{cases} \tag{16}$$

which belongs to $\mathcal{V}_\delta$ by construction. We can assume now that $\mathcal{R}_\delta$ is not empty.
$\Rightarrow$) Assume $\mathcal{V}_\delta \neq \emptyset$ and let $V \in \mathcal{V}_\delta$. For all $R \in \mathcal{R}_\delta$, $\delta(b_{\langle R\rangle}) = 1$ by Def. 7.2, then $on_V(\langle R\rangle)$. Therefore $V \approx\!\!\models R$ by Eq. (14) and consequently R_δ is satisfiable.
$\Leftarrow$) If R_δ is satisfiable then there exist values v_x for each variable $x \in fv(R_\delta)$ which satisfy R_δ. On their basis we define $W \in \mathcal{V}_A$ (with v an arbitrary value) by:

$$W(x) = \begin{cases} \star & \text{if } \delta(b_{\widehat{x}}) = 0 \\ v_x & \text{if } x \in fv(R_\delta) \\ v & \text{for the remaining cases.} \end{cases} \tag{17}$$

Notice that W is well defined since by Lemma 7.1, the three cases of Eq. (17) are disjoint. We prove now that $W \in \mathcal{V}_\delta$, that is, for all $h \in BCT(\phi)$,

$$on_W(h) \text{ iff } \delta(h_B) = 1 \tag{18}$$

By construction of W, (18) holds for clock terms $\mathbb{O}$ and $\widehat{x}$ ($x \in A$). Therefore

$$on_W(\Upsilon(R)) \text{ iff } \delta(\Upsilon(R)_B) = 1, \text{ for all } \langle R\rangle \in BCT(\phi) \tag{19}$$

We now show (18) for all $\langle R\rangle \in BCT(\phi)$. Let $\langle R\rangle \in BCT(\phi)$.
If $\delta(b_{\langle R\rangle}) = 1$, then $R \in \mathcal{R}_\delta$ by Def. 7.2, hence $\delta(\Upsilon(R)_B) = 1$ by Lem. 7.1. As moreover $W \approx\!\!\models R$ by construction, by Eq. (14) and (19) we have $on_W(\langle R\rangle)$. Otherwise, $\delta(b_{\langle R\rangle}) = 0$. Two cases can be distinguished:

- if $\delta(\Upsilon(R)) = 0$, then By Eq. (19) *not* $on_W(\Upsilon(R))$ necessary holds, which in turn entails *not* $on_W(\langle R\rangle)$ by Eq. (14).
- if $\delta(\Upsilon(R)) = 1$, then by Eq. (15) we necessarily have $\delta(b_{\langle \neg R\rangle}) = 1$, therefore $\neg R \in \mathcal{R}_\delta$. By the first case of the proof, we have $on_W(\langle \neg R\rangle)$ which implies by Eq. (14) that $W \approx\!\!\models \neg R$. Hence $W \not\approx\!\!\models R$ and therefore *not* $on_W(\langle R\rangle)$.

□

Notice that assuming that ϕ is a completed formula is necessary to get Th. 7.2: consider the non-completed formula $\langle x > 0\rangle \subseteq \widehat{x} \ \wedge \ \langle \neg(x > 0)\rangle \subseteq \widehat{x}$. Define the distribution $\delta(b_{\widehat{x}}) = 1$, and 0 otherwise, in particular $\delta(b_{\langle x>0\rangle}) = 0$ and $\delta(b_{\langle \neg(x>0)\rangle}) = 0$. Although δ is a model of ϕ_B, it cannot be concretized into a valuation, entailing emptiness for $\mathcal{V}_\delta$. And yet $\mathcal{R}_\delta$ is also empty.

Corollary 7.2 *ϕ is satisfiable iff there exists $\delta \in \Delta$ s.t. $\delta \models \phi_B$ and R_δ is satisfiable.*

Proof $\Leftarrow$) Assume there exists δ s.t. $\delta \models \phi_B$ and R_δ is satisfiable. By Th. 7.2 there exists $V \in \mathcal{V}_\delta$, and by Cor. 7.1 $V \models \phi$.
$\Rightarrow$) Assume ϕ is satisfiable and let $V \models \phi$. By Cor. 7.1 there exists $\delta \in \Delta$ s.t. $\delta \models \phi_B$ and $V \in \mathcal{V}_\delta$. So R_δ is satisfiable by Th. 7.2. □

7.2 The Decision Procedure

Provided the first order language $\mathcal{L}$ is decidable, Cor. 7.2 induces a decision procedure for $\mathcal{CL}$. The algorithm of Fig. 24 takes $\phi \in \mathcal{CF}^{qf}$ as input, and returns YES if ϕ is satisfied by a valuation $V \neq {}^\star V_A$. Otherwise it returns NO. Note that a quantified formula $\exists x.\phi$ is satisfiable iff ϕ is satisfiable.

```
Let DIST = {δ ∈ Δ | δ ⊨ φ_B and δ ≠ 0} in
For all δ ∈ DIST do
  if R_δ = ∅ then return YES else
     if R_δ is satisfiable then return YES;
return NO.
```

Fig. 24: A decision procedure for the satisfiability of ϕ

Because the set $DIST$ is finite, the algorithm terminates. It is correct by Cor. 7.2. The decision procedure above can be improved in order to effectively exhibit a model of ϕ, provided a constructive decision procedure exists for $\mathcal{L}$: the definition of the valuation W by Eq. (16) and Eq. (17) returns a model of ϕ.

Complexity issues can be briefly answered as follows. Define $|\phi|$ as the number of all symbols occurring in ϕ, included the symbols that appear in formulas $R \in \mathcal{F}(\mathcal{L})$ for $\langle R\rangle \in BCT(\phi)$. First, solutions of ϕ_B are enumerated. This runs in exponential time on $|\phi_B|$ ($\leq |\phi|$), since checking that a distribution satisfies ϕ_B is linear time. Hence, this enumeration is in $O(2^{|\phi|})$. Then, for each distribution δ, some decision procedure of $\mathcal{L}$ is performed for R_δ; notice that $|R_\delta| \leq |\phi|$. By calling $\mathbf{C}_\mathcal{L}(|R|)$ the complexity for deciding $R \in \mathcal{F}(\mathcal{L})$, we obtain an upper bound complexity for the decision algorithm of $\mathcal{CL}$ in $O(2^{|\phi|} * \mathbf{C}_\mathcal{L}(|\phi|))$. For the lower bound, the complexity is the maximum of (1) the lower bound for enumerating all solutions of a propositional formula (at least NP-hard because of SAT problem) and (2) the complexity of the decision for $\mathcal{L}$.

8 Model-checking for Data-flow Specifications

In this section we derive from the decision procedure a model-checking algorithm for SIGNAL programs. We show in Sect. 8.1 how the static abstraction of programs translates into clock formulas, then model-checking is discussed in Sect. 8.2.

8.1 Translation of $\mathcal{S}$SIGNAL into the $\mathcal{CL}$ Language

We give in Fig. 25 a mapping $T_{\mathcal{CL}}$ which associates to a $\mathcal{S}$SIGNAL program `P` a clock formula $T_{\mathcal{CL}}(\mathtt{P}) \in \mathcal{CF}$. This translation is correct thanks to Th. 8.1. Note that synchronizations translate directly into clock formulas, e.g. `x ^= y` translates into $\widehat{x} \equiv \widehat{y}$.

Theorem 8.1 *For all $\mathcal{S}$*SIGNAL *program `P` and $V \in \mathcal{V}_S$,*

$$V \in [\![P]\!]_{\mathcal{V}_S} \text{ iff } V \models T_{\mathcal{CL}}(P).$$

Proof The proof is by induction. Let us consider $V \in \mathcal{V}_S$.
Basic Case:
• `P = y := g(x1, ... ,xn)`: by trivial rewriting of the SIGNAL flow semantics given on Sect. 3.1 we have:

$\mathcal{S}$SIGNAL program P	$\mathbf{T}_{\mathcal{CL}}(\mathtt{P}) \in \mathcal{CF}$
`y := g(x1, ... ,xn)`	$\widehat{y} \equiv \widehat{x_1} \equiv \ldots \equiv \widehat{x_n} \ \wedge \ \widehat{y} \subseteq \langle y = g(x_1, \ldots, x_n) \rangle$
`y := x when c`	$\widehat{y} \equiv \widehat{x} \cap \langle c \rangle \ \wedge \ \widehat{y} \subseteq \langle y = x \rangle$
`y := u default v`	$\widehat{y} \equiv \widehat{u} \cup \widehat{v} \ \wedge \ \widehat{u} \subseteq \langle y = u \rangle \ \wedge \ \widehat{v} \setminus \widehat{u} \subseteq \langle y = v \rangle$
`( P )/x`	$\exists x. T_{\mathcal{CL}}(\mathtt{P})$
`P1 | P2`	$T_{\mathcal{CL}}(\mathtt{P_1}) \wedge T_{\mathcal{CL}}(\mathtt{P_2})$

Fig. 25: $\mathcal{CL}$ semantics for $\mathcal{S}$SIGNAL

$$\begin{aligned} V \in [\![\mathtt{P}]\!]_{\mathcal{V}_S} \text{ iff } & V(y) = \star \Rightarrow V(x_1) = \cdots = V(x_n) = \star \\ & \wedge\ V(y) \neq \star \Rightarrow \ \forall i, V(x_i) \neq \star \wedge V(y) = g(V(x_1), \ldots, V(x_n)) \\ \text{iff } & on_V(y) \Leftrightarrow on_V(x_1) \Leftrightarrow \ldots \Leftrightarrow on_V(x_n) \\ & \wedge\ on_V(y) \Rightarrow \ \forall i, on_V(x_i) \wedge V \approx y = g(x_1, \ldots, x_n) \\ \text{iff } & V \models \widehat{y} \equiv \widehat{x_1} \equiv \ldots \equiv \widehat{x_n} \ \wedge \ \widehat{y} \subseteq \langle y = g(x_1, \ldots, x_n) \rangle. \end{aligned}$$

• $\mathtt{P} =$ `y := x when c`: the simplest proof is to make explicit the set $[\![\mathtt{P}]\!]_{\mathcal{V}_S}$. T stands for *true* and F for *false*. $V \in [\![$ `y := x when c` $]\!]_{\mathcal{V}_S}$ iff V satisfies:

$$(V(c) = T \wedge V(y) = \star \wedge V(x) = \star) \ \bigvee \ (V(c) = T \wedge V(y) \neq \star \wedge V(y) = V(x)) \ \bigvee \ (V(c) \neq T \wedge V(y) = \star)$$

$$\begin{aligned} \text{iff} \quad & (V(y) \neq \star \ \Leftrightarrow \ V(c) = T \wedge V(x) \neq \star) \bigwedge (V(y) \neq \star \Rightarrow V(y) = V(x)) \\ \text{iff} \quad & (on_V(y) \ \Leftrightarrow \ on_V(\langle c \rangle) \wedge on_V(x)) \bigwedge (on_V(y) \ \Rightarrow \ on_V(x) \wedge V \approx x = y) \\ \text{iff} \quad & V \models \widehat{y} \equiv \widehat{x} \cap \langle c \rangle \ \bigwedge \ \widehat{y} \subseteq \langle y = x \rangle. \end{aligned}$$

• $\mathtt{P} =$ `y := u default v`: similarly $V \in [\![\mathtt{P}]\!]_{\mathcal{V}_S}$ iff V satisfies

$$(V(u) \neq \star \wedge V(y) = V(u)) \ \bigvee \ (V(u) = \star \wedge V(v) \neq \star \wedge V(y) = V(v)) \ \bigvee \ (V(u) = V(v) = V(y) = \star)$$

$$\begin{aligned} & \text{iff } (V(y) \neq \star \Leftrightarrow V(u) \neq \star \vee V(v) \neq \star) \ \bigwedge \ (V(u) \neq \star \Rightarrow V(y) = V(u)) \ \bigwedge \ (V(u) = \star \Rightarrow V(y) = V(v)) \\ & \text{iff } V \models \widehat{y} \equiv \widehat{u} \cup \widehat{v} \ \wedge \ \widehat{u} \subseteq \langle y = u \rangle \ \wedge \ \widehat{v} \setminus \widehat{u} \subseteq \langle y = v \rangle. \end{aligned}$$

Inductive Step:

• $\mathtt{P} =$ `( P1 )/x` where $\mathtt{P_1} \in \mathcal{P}g_{A_1}$:

$$\begin{aligned} V \in [\![\mathtt{P}]\!]_{\mathcal{V}_S} \text{ iff } & V \in [\![\textstyle\prod_{A \setminus \{x\}} ([\![\mathtt{P_1}]\!]) \,]\!]_{\mathcal{V}_S} \\ & \text{iff } \exists V_1 \in [\![\mathtt{P_1}]\!]_{\mathcal{V}_S}.\ V_{1|A\setminus\{x\}} = V_{|A\setminus\{x\}} \\ & \text{iff } \exists V_1 \in \mathcal{V}_{A_1}.\ V_1 \models T_{\mathcal{CL}}(\mathtt{P_1}) \wedge V_{1|A\setminus\{x\}} = V_{|A\setminus\{x\}} \text{ by induction hypothesis} \\ & \text{iff } V \models \exists x. T_{\mathcal{CL}}(\mathtt{P_1}). \end{aligned}$$

• $\mathtt{P} = \mathtt{P_1} \mid \mathtt{P_2}$ where $\mathtt{P_1} \in \mathcal{P}g_{A_1}$ and $\mathtt{P_2} \in \mathcal{P}g_{A_2}$:

$$\begin{aligned} V \in [\![\mathtt{P}]\!]_{\mathcal{V}_S} \text{ iff } & V \in [\![\, [\![\mathtt{P_1}]\!] \parallel [\![\mathtt{P_2}]\!] \,]\!]_{\mathcal{V}_S} \\ & \text{iff } V_{|A_1} \in [\![\mathtt{P_1}]\!]_{\mathcal{V}_S} \wedge V_{|A_2} \in [\![\mathtt{P_2}]\!]_{\mathcal{V}_S} \\ & \text{iff } V_{|A_1} \models T_{\mathcal{CL}}(\mathtt{P_1}) \wedge V_{|A_2} \models T_{\mathcal{CL}}(\mathtt{P_2}) \text{ by induction hypothesis} \\ & \text{iff } V \models T_{\mathcal{CL}}(\mathtt{P_1}) \wedge V \models T_{\mathcal{CL}}(\mathtt{P_2}) \text{ since } \forall x \in A_1 \cap A_2, V_{|A_1}(x) = V_{|A_2}(x) \\ & \text{iff } V \models T_{\mathcal{CL}}(\mathtt{P_1} \mid \mathtt{P_2}). \end{aligned}$$

□

Two kinds of clock formulas appear in $T_{\mathcal{CL}}(\mathtt{P})$. On the one hand we recognize relations between clocks induced by the abstraction by synchronizations presented in Sect. 5.1.4 (e.g. $\widehat{y} \equiv \widehat{x} \cap \langle c \rangle$). These relations represent the control part of the program $\mathtt{P}_{\mathcal{C}}$. On the other hand, its data part $\mathtt{P}_{\mathcal{D}}$ appears through clock formulas of the kind $h \subseteq \langle R \rangle$. The formula $R \in \mathcal{F}(\mathcal{L})$ represents the relation between variables induced by a definition *activated* by h (a definition is said to be activated by a clock h if the induced assignment is performed at all instants of h). For example y is always defined by x ($\widehat{y} \subseteq \langle y = x \rangle$) and y is defined by v at instants of $\widehat{v} \setminus \widehat{u}$ ($\widehat{v} \setminus \widehat{u} \subseteq \langle y = v \rangle$). It corresponds to the intuition that h is a guard for the activation of the definition ("R holds at instants of h").

Note that not all properties expressed by $\mathcal{CL}$ can be expressed in SIGNAL, which does not provide negation. As a consequence SIGNAL can express neither that a clock is not empty, nor that a clock inclusion is strict.

Remark 8.1 The inclusion $\widehat{y} \subseteq \langle y = g(x_1, \ldots, x_n)\rangle$ that appears in $T_{\mathcal{CL}}(\mathtt{P})$ in the case of an instantaneous function is indeed an equality: $\widehat{y} \equiv \langle y = g(x_1, \ldots, x_n)\rangle$. As a matter of fact, $\Upsilon(y = g(x_1, \ldots, x_n)) = \widehat{y} \cap \widehat{x_1} \cap \cdots \cap \widehat{x_n}$ and trivially $\models \widehat{y} \cap \widehat{x_1} \cap \cdots \cap \widehat{x_n} \subseteq \widehat{y}$, therefore $\models \langle y = g(x_1, \ldots, x_n)\rangle \subseteq \widehat{y}$. It is also the case for inclusions $\widehat{y} \subseteq \langle y = x\rangle$ (in the case of a filtering) and $\widehat{u} \subseteq \langle y = u\rangle$ (in the case of a merge). On the contrary, the inclusion $\widehat{v} \setminus \widehat{u} \subseteq \langle y = v\rangle$ induced by the second part of the `default` operator is not an equality: it might be the case that u and v are present at the same time with the same values. Then for such a valuation $V \in \mathcal{V}_S$, *not* $on_V(\widehat{v} \setminus \widehat{u})$ but $on_V(\langle y = v\rangle)$.

Example 8.1 We give on Fig. 26 the clock formulas ϕ_{abs} which represents the absolute value of Ex. 3.2 and ϕ_{ct} which represents the counter of Ex. 3.1. ◇

$$\begin{array}{l} \widehat{p} \equiv \langle y \geq 0\rangle \equiv \langle p = y\rangle \equiv \langle a = p\rangle \\ \wedge\ \widehat{n} \equiv \langle y < 0\rangle \equiv \langle n = -y\rangle \\ \wedge\ \widehat{n} \setminus \widehat{p} \subseteq \langle a = n\rangle \\ \wedge\ \widehat{a} \equiv \widehat{p} \cup \widehat{n} \end{array}$$

(a) ϕ_{abs}

$$\begin{array}{l} \widehat{a} \equiv \widehat{ma} \equiv \langle a = N\rangle \cup \widehat{a} \\ \wedge\ \langle ma = 0\rangle \equiv \langle a = N\rangle \equiv \widehat{N} \equiv \langle N \geq 0\rangle \\ \wedge\ \widehat{ma} \setminus \widehat{N} \subseteq \langle a = ma - 1\rangle \end{array}$$

(b) ϕ_{ct}

Fig. 26: Clock Formulas for Ex. 3.1 and 3.2

8.2 Model-checking

We show here how to check whether a SIGNAL program P satisfies an instantaneous property $\phi \in \mathcal{CF}^{qf}$, thanks to the decision procedure. We aim at answering the problem "does any flow $F \in [\![\mathtt{P}]\!]$ satisfies ϕ ?", which reduces to "does any valuation $V \in [\![\mathtt{P}]\!]_{\mathcal{V}_S}$ satisfies ϕ ?", and in turn, by using the translation from $\mathcal{S}$SIGNAL to $\mathcal{CL}$, to "$\models T_{\mathcal{CL}}(\mathtt{P}_{\mathcal{S}}) \Rightarrow \phi$?". The decision procedure for $\mathcal{CL}$ is then used to answer the satisfiability of $T_{\mathcal{CL}}(\mathtt{P}_{\mathcal{S}}) \wedge \neg\phi$ (in which quantifiers are pushed out, e.g. by renaming techniques). As expected, since an abstraction of P is used, if the answer is "NO" then P satisfies ϕ, otherwise nothing can be inferred.

Example 8.2 Let us consider the absolute value given on Ex. 3.2 and prove that "a is always positive". This instantaneous property can be expressed by ϕ: $\widehat{a} \subseteq \langle a \geq 0\rangle$. Let us denote by ϕ' the completion of $\phi_{abs} \wedge \neg\phi$. Three distributions δ_1, δ_2, δ_3 satisfy ϕ'_B. They differ depending on whether they "switch on" $\widehat{p}$ (resp. $\widehat{n}$) or not. We show that none of them corresponds to a concrete valuation. Say δ_1 switches on clocks $\widehat{p}$ and $\widehat{n}$, then also $\langle y \geq 0\rangle$ and $\langle y < 0\rangle$. Therefore the exclusive conditions $y \geq 0$ and $y < 0$ belong to $\mathcal{R}_{\delta_1}$, then R_{δ_1} is not satisfiable. Hence $\mathcal{V}_{\delta_1} = \emptyset$. δ_2 switches off $\widehat{n}$ and switches on $\widehat{p}$, hence also $\langle y \geq 0\rangle$, $\langle p = y\rangle$, $\langle a = p\rangle$ (see ϕ_{abs}) and $\langle \neg y < 0\rangle$ and $\langle \neg a \geq 0\rangle$ (due to completion); therefore R_{δ_2} is not satisfiable, hence $\mathcal{V}_{\delta_2} = \emptyset$. The reasoning is symmetrical for δ_3 which switches on $\widehat{n}$ and switches off $\widehat{p}$. Thus the property is proved. ◇

Example 8.3 Let us consider the counter given on Ex. 3.1. It is easy to prove like in Ex. 8.2 that "N is always positive". Similarly we prove the following inductive property: it is always true that $0 \leq ma \leq N$. Initially, it is verified since $ma = 0$. We then prove the induction step, that is if $0 \leq ma \leq N$ then $0 \leq a \leq N$ (recall that ma takes the past value of a). It is expressed by the clock formula ϕ: $\langle 0 \leq ma \leq N\rangle \subseteq \langle 0 \leq a \leq N\rangle$. Let us denote by ϕ' the completion of $\phi_{ct} \wedge \neg\phi$. δ_1, δ_2, δ_3 satisfy ϕ'_B. Both switch on $\widehat{a}$, $\widehat{ma}$, $\langle 0 \leq ma \leq N\rangle$ and $\langle \neg 0 \leq a \leq N\rangle$ (due to completion). δ_1 and δ_2 also switch on $\langle ma = 0\rangle$ and $\langle a = N\rangle$ (re-initialization of the counter) therefore R_{δ_1} and R_{δ_2} are not satisfiable. δ_3 corresponds to decrementation of the counter and switches on $\langle a = ma - 1\rangle$ and $\langle \neg ma = 0\rangle$ (due to completion), therefore R_{δ_3} entails the formula $\neg 0 \leq a \leq N \wedge 0 \leq a < N$ thus is not satisfiable. Thus the property is proved. ◇

Most of the reactive systems specified in SIGNAL use multiple theories: in this case $\mathcal{L}$ is multi-sorted and the formula R_δ must be split into sub-formulas to perform separate satisfiability tests. On the other hand a program does not use sophisticated theories for non-boolean variables, but e.g. the Presburger algebra or the additive theory of reals. In particular relations between variables do not contain any quantifier.

9 Conclusion and Perspectives

The present work defines a language $\mathcal{CL}$ based on clock relations, which sentences (or formulas) are naturally suitable for the synchronous paradigm. The $\mathcal{CL}$ language enables to express some safety properties, which describe something happening inside a single reaction. In particular, as expected, the static abstraction of synchronous data-flow specifications can be translated into $\mathcal{CL}$. The decidability of $\mathcal{CL}$ (i.e. the existence of an algorithm deciding whether a formula is satisfied by a non-trivial valuation) is proved, provided $\mathcal{CL}$ constructs (clocks) rely on a decidable theory. The decision procedure is based on a boolean abstraction of $\mathcal{CL}$ formulas into propositional calculus formulas: it first determines whether a boolean distribution is a model of the abstract formula, and next if it can be made a concrete valuation from this distribution. Finally, model-checking is derived for SIGNAL programs and illustrated by two toy examples.

In the five first sections of the document we go back over abstractions used to analyze SIGNAL programs. Their link with classical abstract-based analyses is highlighed, in particular the concept of structural abstraction is introduced and related to abstract interpretation. The way absence is handled is emphasized. It is shown that analyzing the admissible valuations of specifications is necessary, and motivations are given for the introduction of the $\mathcal{CL}$ language.

9.1 Related Work

We mainly focus on analyses related to SIGNAL, then we briefly consider LUSTRE.

9.1.1 Analyses related to SIGNAL

The Clock Algebra used to encode the structural static abstraction and the abstraction by synchronizations of SIGNAL programs is a clear precursor of the $\mathcal{CL}$ language. Nevertheless it is isomorphic to the propositional calculus, and deals only with boolean aspects of programs. Our $\mathcal{CL}$ induces a strict increase of expressiveness since relations between non-boolean variables are also taken into account. As shown in Sect. 8.2, their consideration is necessary to prove some properties.

A recent work [22] prepares the practical interaction of clocks and values of variables in the particular case of the abstract domain of intervals. [22] defines i-formulas, that are formulas of the propositional calculus extended with predicates of the kind $x \in I$ (represented by a pair (x, I)), for $x \in S$ and $I \in \mathcal{I}$. Existential and universal quantifications of i-formulas over variables in S are also provided. The propositional calculus is extended with rules that apply to predicates of the kind $x \in I$ and a canonical form for i-formulas is given. [22] explains how i-formulas are implemented by IBDD, that are an extension of classical BDD with variables representing predicates (x, I) (these variables are upper nodes of IBDD, leaves being BDD). The choice of BDD authorizes a direct connection with the clock calculus. The representation of abstract programs by means of clocks and predicates of the kind $x \in I$ inserts naturally into the $\mathcal{CL}$ language as a particular case: the relation-clock $\langle x \in I \rangle$ can be defined as follows:

$$on_V(\langle x \in I \rangle) \text{ iff } V(x) \neq \star \text{ and } V(x) \in I$$

There is no need of function Υ since an interval states a property on one variable and not a relation between several ones. An analysis of SIGNAL programs by means of intervals is in course of definition.

The only other analysis that deals with values of non-boolean variables is the one of [11] mentioned in Sect. 3.2.2 and 5.3. The normalization stage handles the static part $\mathtt{P}_S$ of programs: it can be seen as a synthesis of instantaneous properties. Nevertheless clocks are not used as objects of the analysis: in practice $\mathtt{P}_S$ is transformed into a set of pairs composed of a list of absent variables and a polyhedron stating a relation between the present ones. There is a clear correspondence between absence of variables indicated by lists and distributions, as well as between a polyhedron and our relation R_δ. Our work therefore can be seen as a clean formalization of the normalization stage. Moreover [11] does not prove that constraints represent exactly the set of admissible valuations of the specification. It is only stated that they contain it. Our formalization is then cleaner for this point.

9.1.2 Analyses related to LUSTRE

LUSTRE is the data-flow language closest to SIGNAL. Numerous analyses and verification tools are dedicated to LUSTRE programs (e.g. Lesar [29] for model-checking, Lurette [50] for test, and recently proofs using PVS [5]). However models used are quite far from clocks, e.g. analyses apply to a control automaton obtained by partial

evaluation of the boolean variables of the program [27]. In particular no significant analysis of the static part of programs is performed during the *clock verification* [16], the analogous of the clock calculus for SIGNAL (one can refer to [35] for a general and formal presentation). As shown by this document, the application of works on the $\mathcal{CL}$ language to LUSTRE is nevertheless theoretically possible.

9.2 Perspectives

The decision procedure we give addresses first boolean aspects and next numerical ones. It proves that a mixing between "boolean" clocks and values is possible, but does not provide a real interaction between the boolean resolution and the numerical one. This interaction is needed in practical algorithms to simulate the interconnection between control and data, as much as possible. It is e.g. needless to perform a boolean reasoning that involves distributions which clearly do not correspond to a concrete model, as well as to perform too early expensive tests for the satisfiability of numerical formulas if no boolean model exists anyway. We currently investigate algorithmic techniques based on the construction of a graph whose nodes are clocks and edges denote the specified inclusion between clocks. Subsets of $\mathcal{F}(\mathcal{L})$ are associated to nodes, in such a way that if R is associated to h then $h \subseteq \langle R \rangle$. Therefore we can e.g. safely deduce from the non-satisfiability of R the emptiness of h.

Once such a structure is built in a bottom-up approach (with an accuracy to be determined), it should be exploited to solve various problems that concern the static part of processes. An immediate application would be the improvement of the clock calculus, leading e.g. to higher quality executable code. As a second application, it would worth modifying the normalization algorithm of [11] (admitted to be naive) to introduce clocks as in our approach, and look if it yields to better results.

Index

References

[1] Pascalin Amagbégnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 163–173, La Jolla, California, 18–21 June 1995.

[2] T. Amagbégnon, L. Besnard, and P. Le Guernic. Arborescent canonical form of boolean expressions. Technical Report 2290, Inria, June 1994.

[3] H. Andersen and H. Hulgaard. Boolean expression diagrams. In *12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, pages 88–98, Washington - Brussels - Tokyo, June 1997. IEEE.

[4] G. Behrmann, K. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification*, pages 341–353, 1999.

[5] S. Bensalem, P. Caspi, C. Dumas, and C. Parent-Vigouroux. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7, San Jose*. IEEE Computer Society, January 1999.

[6] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Conference on Computer Aided Verification CAV'98*, LNCS 1427, pages 319–331, 1998.

[7] A. Benveniste and G. Berry. The synchronous approach to reactive and real–time systems. *IEEE Trans. Autom. Control*, 9(79):1270–1282, September 1991.

[8] A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 1992.

[9] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[10] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[11] F. Besson, T. Jensen, and J.P. Talpin. Polyhedral analysis for synchronous languages. In A. Cortesi and G. Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 1999.

[12] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, faculté de sciences appliquées, 1998.

[13] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state programs using presburger arithmetic. Technical report, University of Maryland Institute for Advanced Computer Studies Dept. of Computer Science, sept 1996.

[14] Tevfik Bultan, Richard Gerber, and William Pugh. Model checking concurrent systems with unbounded integer variables: Symbolic representations, approximations and experimental results. Technical report, University of Maryland Institute for Advanced Computer Studies Dept. of Computer Science, feb 1998.

[15] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[16] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proc. of 14th ACM Conf. on Principles of Programming Languages*, pages 178–188. ACM Press, 1987.

[17] SACRES Consortium. The semantic foundations of SACRES. Technical report, Esprit Project EP 20897: Safety Critical Embedded Systems, March 1997.

[18] René Cori and Daniel Lascar. *Logique mathématique, cours et exercices*, volume 1. Masson, 1993.

[19] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.

[20] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[21] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *5th ACM Symposium on Principles of Programming Languages*, 1978.

[22] Abdoulaye Gamatié. Abstraction de domaines totalement ordonnés et calcul d'horloge. Rapport de DEA, IFSIC, Université de Rennes 1, 2000. In french.

[23] T. Gautier, P. Le Guernic, and F. Dupont. Signal v4 : manuel de référence. Technical Report 832, Irisa, 1994.

[24] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[25] P. Le Guernic and Thierry Gautier. *Advanced Topics in Data-Flow Computing*, chapter Data-Flow to von Neumann: the Signal approach. JL. Gaudiot, 1991.

[26] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[27] N. Halbwachs. About synchronous programming and abstract interpretation. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur (belgium), September 1994. LNCS 864, Springer Verlag.

[28] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[29] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[30] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *International Static Analysis Symposium, SAS'94*, Namur (Belgium), September 1994.

[31] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[32] M. Handjieva. Abstract interpretation of constraint logic programs using convex polyhedra. Technical report, LIX, Ecole Polytechnique, 1996. URL http://lix.polytechnique.fr/~handjiev/HANDJIEVpapers.html.

[33] David Harel and Amnon Naamad. The STATEMATE Semantics of Satecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[34] B. Jeannet. *Dynamic partitionning in linear relation analysis and application to the verification of synchronous programs*. PhD thesis, Institut National Polytechnique de Grenoble, 2000. in French.

[35] Thomas P. Jensen. Clock analysis of synchronous dataflow programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 156–167, La Jolla, California, 21-23 June 1995.

[36] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[37] Richard Lassaigne and Michel De Rougemont. *Logique et fondements de l'informatique*. Hermès, 1993.

[38] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of signal programs: Application to a power transformer station controller. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology AMAST'96*, pages 271–285, Munich, Germany, July 1996. Springer-Verlag, LNCS 1101.

[39] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design Volume 6, Issue 1*, 1995.

[40] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[41] H. Marchand and M. Samaan. Incremental design of a power transformer station controller using controller synthesis methodology. *IEEE Transaction on Software Engineering*, 26(8):729–741, august 2000.

[42] C. Mauras. Calcul symbolique et automates interprétés. Technical Report 96.10, Laboratoire d'Automatique de Nantes LAN, nov 1996.

[43] J. Møller, J. Lichtenberg, H. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, 1999.

[44] D. Nowak. *Spécification et preuve de systèmes réactifs*. PhD thesis, Université de Rennes 1, IFSIC, 1999.

[45] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[46] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.

[47] Amir Pnueli, N. Shankar, and Eli Singerman. Fair synchronous transition systems and their liveness proofs. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 198–209, Lyngby, Denmark, September 1998. Springer-Verlag.

[48] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In IEEE, editor, *Proceedings, Supercomputing '91: Albuquerque, New Mexico, November 18–22, 1991*, pages 4–13. IEEE Computer Society Press, 1991.

[49] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, 1 May 1998.

[50] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[51] R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[52] V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99*, 1998.

[53] I. Smarandache. *Transformations affines d'horloges: application au codesign de systèmes temps-réel en utilisant les langages Signal et Alpha*. PhD thesis, Université de Rennes 1, IFSIC, 1998. In french.

[54] Karsten Strehl. Using interval diagram techniques for the symbolic verification of timed automata. Technical Report 53, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, July 1998.

[55] Karsten Strehl and Lothar Thiele. Interval diagram techniques and their applications. In *Proceedings of the 8th International Workshop on Post-Binary ULSI Systems*, pages 23–24, Freiburg im Breisgau, Germany, May 19, 1999. Invited paper.

[56] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[57] J.-P. Talpin, A. Benveniste, B. Caillaud, and P. Le Guernic. Hierarchic normal forms for desynchronization. Technical Report 1288, Irisa, December 1999.

[58] Williams, Andersen, and Hulgaard. Satisfiability checking using boolean expression diagrams. In *TACAS: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, 2001.

[59] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 1–19, 2000.