Modular interpretation of heterogeneous modeling diagrams into synchronous equations using static single assignment

Jean-Pierre Talpin, Julien Ouy, Thierry Gautier, Loïc Besnard, Cortier Alexandre

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modular interpretation of heterogeneous modeling diagrams into synchronous equations using static single assignment

Jean-Pierre Talpin, Julien Ouy, Thierry Gautier, INRIA

Loic Besnard, CNRS, and Alexandre Cortier, CNES-IRIT

N° 7036

September 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7036--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Modular interpretation of heterogeneous modeling diagrams into synchronous equations using static single assignment

Jean-Pierre Talpin, Julien Ouy, Thierry Gautier, INRIA
Loic Besnard, CNRS, and Alexandre Cortier, CNES-IRIT

Thème COM — Systèmes communicants
Équipes-Projets Espresso

Rapport de recherche n° 7036 — September 2009 — 19 pages

Abstract: The ANR project SPACIFY develops a domain-specific programming environment, Synoptic, to engineer embedded software for space applications. Synoptic is an Eclipse-based modeling environment which supports all aspects of aerospace software design. As such, it is a domain-specific environment consisting of heterogeneous modeling and programming principles defined in collaboration with the industrial partners and end users of the project : imperative synchronous programs, data-flow diagrams, mode automata, blocks, components, scheduling, mapping and timing. This article focuses on the essence and distinctive features of its behavioral or programming aspects : actions, flows and automata, for which we use the code generation infrastructure of the synchronous modeling environment SME. It introduces an efficient method for transforming a hierarchy of blocks consisting of actions (sequential Esterel-like programs), data-flow diagrams (to connect and time modules) and mode automata (to schedule or mode blocks) into a set of synchronous equations. This transformation minimizes the needed state variables and block synchronizations. It consists of an inductive static-single assignment transformation algorithm across a hierarchy of blocks that produces synchronous equations. The impact of this new transformation technique is twofold. With regards to code generation objectives, it minimizes the needed resynchronization of each block in the system with respects to its parents, potentially gaining substantial performance from way less synchronizations. With regards to verification requirements, it minimizes the number of state variables across a hierarchy of automata and hence maximizes model checking performances.

Key-words: Synchronous programming, model-driven engineering, program transformation, static-single assignment

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Interpretation modulaire de modèles hétérogènes utilisant l'assignation unique

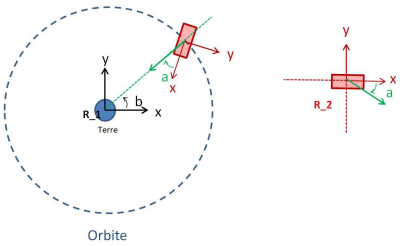
Résumé : Ce rapport décrit une technique de transformation de programme utilisée dans le cadre du projet ANR Spacify afin de transformer un ensemble de diagrammes hétérogènes, représentant différentes vue d'un système pendant sa conception, en utilisant une représentation intermédiaire SSA (static-signgle assignment). La forme intermdiaire, modulaire et compacte, ainsi obtenue, est alors interprété dans lem odèle de calcul synchrone de l'atelier SME pour obtenir enfin une modélisation formelle supportant une vérification accéléré et un code gńéré efficace

Mots-clés : Programmation synchrone, ingénierie de modèles, transformation de programmes, assignation unique

Contents

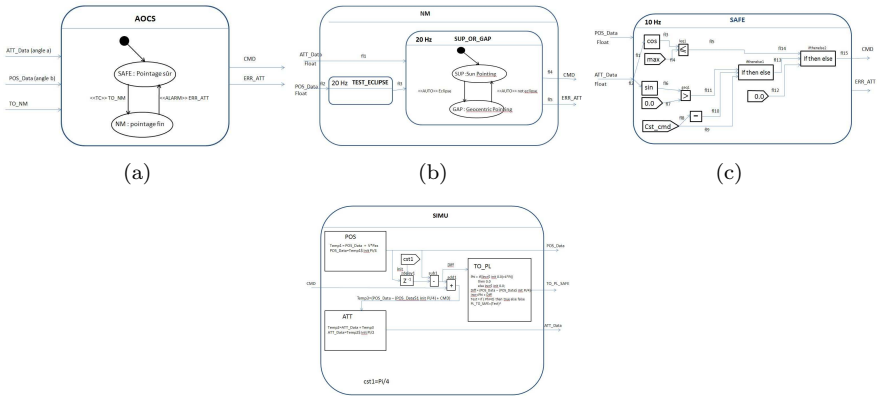
1 Introduction

In the ANR project SPACIFY [10], we develop a domain-specific programming environment, Synoptic, for engineering for space application and control software. Synoptic is an Eclipse-based modeling environment which supports all aspects of aerospace software design. It covers the design of application and control modules using imperative synchronous programs, data-flow diagrams, mode automata, and also the partitioning, timing and mapping of these module onto satellite architectures. As such, Synoptic is a domain-specific environment : its aim is to provide the engineer with a unified modeling environment to cover all heterogeneous programming, analysis and verification tasks, as defined in collaboration with the industrial end users of the project. One typical case study under investigation in the project is a generic satellite positioning software, responsible for automatically moving the satellite into a correct position before initiating interaction with the ground.



Its specification, Fig. 1, consists of the composition of heterogeneous diagrams. Each diagram represents one specfic aspect of the software's role: automata, Fig. 1-1, control its modes of operation for specific conditions (e.g. a solar eclipse). Data-flows, Fig. 1, define communication links and/or periodic processing tasks. Timed imperative programs, Fig. 1, specify sequential algorithmic behaviors.

Figure 1: Specification of the positionning software



1.1 Motivation

In this article, we are concerned with the heterogeneity of modeling notations of the Synoptic language and propose a method to embed them in a suitable model of computation for the purpose of formal verification and code generation. To model and compile all distinctive programming

features of Synoptic : imperative programs, data-flows and automata, we use the code generation infrastructure of the synchronous modeling environment SME [2].

This model transformation or interpretation introduces an efficient method for transforming a hierarchy of blocks consisting of actions (sequential Esterel-like programs), data-flow diagrams (to connect and time modules) and mode automata (to schedule or mode blocks) into a set of synchronous equations. This transformation minimizes the needed state variables and block synchronizations. It consists of an inductive static-single assignment transformation algorithm across a hierarchy of blocks that produces synchronous equations.

1.2 Outline

The impact of this transformation technique is twofold. With regard to code generation objectives, it maximizes the amount of atomic operations that are performed within one cycle of execution, gaining substantial performance from way less synchronizations. With regard to verification requirements, it minimizes the number and updates of state variables across automata and hence provides better model checking performances.

Example The principle of our transformation technique can be illustrated by considering a simple (Esterel-like) imperative Synoptic program. When the program receives control, it first initializes x and then increments it if y is true. Otherwise, x is decremented, a skip is issued (to synchronize with the parent block), and x is decremented again. At the end, control is released to the parent (caller) block.

$$
\begin{array}{l}
x = 0; \\
\mathsf{if}\, y \;\; \mathsf{then}\, \{x = x + 1\} \\
\qquad \mathsf{else}\, \{x = x - 1;\, \mathsf{skip}\,;\, x = x - 1\} \\
\mathsf{end}
\end{array}
$$

The static single assignment form of this program (Fig. 2, left) assigns a different name $x_{1\ldots4}$ to each definition of x and uses a so called ϕ-node to merge the values x_2 and x_4 of x flowing from then and else branches of the if. It is interpreted (Fig. 2, right) by a merge (the default keyword) of two flows, x_2 and x_4, that are sampled (the when keyword) by the corresponding condition of the control-flow. All other data-flow equations match an instruction of the intermediate SSA form and its control-flow point. The signals s and s' record the current and next states of the program (0 and 1) using a delay keyword pre.

Figure 2: A timed sequential program and its data-flow interpretation

$x_1 = 0;$	$x_1 = 0 \,\mathsf{when}\, (s = 0)$
$\mathsf{if}\, y$	$x_2 = x_1 + 1 \,\mathsf{when}\, (s = 0) \,\mathsf{when}\, y$
$\quad \mathsf{then}\, x_2 = x_1 + 1$	$x_3 = x_1 - 1 \,\mathsf{when}\, (s = 0) \,\mathsf{when\ not}\, y$
$\quad \mathsf{else}\, \{$	$x_4 = (x \,\mathsf{pre}\, 0) - 1 \,\mathsf{when}\, (s = 1)$
$\qquad x_3 = x_1 - 1;$	$x = \;\; x_2 \,\mathsf{when}\, (s = 0) \,\mathsf{when}\, y$
$\qquad x = x_3;$	$\qquad\qquad \mathsf{default}\, x_3 \,\mathsf{when}\, (s = 0) \,\mathsf{when\ not}\, y$
$\qquad \mathsf{skip}\,;$	$\qquad\qquad \mathsf{default}\, x_4 \,\mathsf{when}\, (s = 1)$
$\qquad x_4 = x - 1$	$s' = \;\; 0 \,\mathsf{when}\, (s = 1)$
$\quad \};$	$\qquad\qquad \mathsf{default}\, 0 \,\mathsf{when}\, (s = 0) \,\mathsf{when}\, y$
$\quad x = \phi(x_2, x_4)$	$\qquad\qquad \mathsf{default}\, 1 \,\mathsf{when}\, (s = 0) \,\mathsf{when\ not}\, y$
end	$s = s' \,\mathsf{pre}\, 0$

In the present paper, we show how to not only translate the core imperative programming features into equation by generalizing the above idea, but also extend it to the mode automata that control the activation of such elementary blocks and to the data-flow diagrams that connect them.

This yields, just as in the case of the above simple programs (one variable update instead of three) a significant gain in the number of transitions and of synchronization needed to verify and to execute programs.

2 Model of computation

The model of computation on which Synoptic relies for program transformation and code generation purposes is that of the Eclipse-based synchronous modeling environment SME [11]. The core of SME is based on compiler of the synchronous programming language Signal [6]. In Signal, a process P consists of the composition of simultaneous equations $x = f(y, z)$ over signals x, y, z.

$$P, Q ::= x = y\, f\, z \mid P/x \mid P \,|\, Q \qquad \text{(process)}$$

A delay equation $x = y\,\mathsf{pre}\,v$ defines x every time y is present. Initially, x is defined by the value v, and then, it is defined by the previous value of y. A sampling equation $x = y\,\mathsf{when}\,z$ defines x by y when z is true. Finally, a merge equation $x = y\,\mathsf{default}\,z$ defines x by y when y is present and by z otherwise. An equation $x = y\,f\,z$ can use a boolean or arithmetic operator f to define all of the n^{th} values of the signal x by the result of the application of f to the n^{th} values of the signals y and z. The synchronous composition of processes $P \,|\, Q$ consists of the simultaneous solution of the equations in P and in Q. It is commutative and associative. The process P/x restricts the signal x to the lexical scope of P.

In Signal, the presence of a value along a signal x is an expression noted $\hat{}x$. It is true when x is present. Otherwise, it is absent. Specific processes and operators are defined in Signal to manipulate clocks explicitly. We only use the simplest one, $x\,\mathsf{sync}\,y$, that synchronizes all occurrences of the signals x and y.

Example We exemplify the model of computation of Signal by considering an equation that defines a counter. The output signal counter is defined by 0 when the input signal reset is true and, otherwise, by an increment of its previous value (counter pre 0) which is initially 0.

$$\mathsf{counter} = 0\,\mathsf{when}\,\mathsf{reset}\,\mathsf{default}\,1 + \mathsf{counter}\,\mathsf{pre}\,0$$

Notice that the clock of the output signal is left deliberately larger than that of the input signal. One may later refine it with an additional clock equation counter sync reset to generate a count every time an occurrence of the input signal reset, true or false, is present.

2.1 Model of timed traces

In the timing model of Polychrony [6], symbolic tags t or u denote periods in time during which execution takes place. Time is defined by a partial order relation $\leq$ on tags: $t \leq u$ stipulates that t occurs before u or at the same time. A chain is a totally ordered set of tags. It corresponds to the clock of a signal: it samples its values over a series of totally related tags. The domains for events, signals, behaviors and processes are defined as follows:

- an *event* is a pair consisting of a tag $t \in \mathbb{T}$ and a value $v \in \mathbb{V}$,
- a *signal* $s \in \mathcal{S}$ is a function from a *chain* of tags $\mathcal{C} \subset \mathbb{T}$ to a set of values $v \in \mathbb{V}$,
- a *behavior* $b \in \mathcal{B}$ is a function from a finite set of signal names $X \subset \mathcal{V}$ to signals,
- a *process* $p \in \mathcal{P}$ is a set of behaviors that have the same domain.

Example To illustrate these definitions, consider a possible behavior of the signals x and y defined by the process $x = \mathsf{filter}(y)$. The time tags at which the output x is present correspond to the time tags at which the previous and present value of y differ. Hence, evenly tagged values belong both to x and y.

$$\begin{array}{lccccccc} y \mapsto & (t_1, \mathsf{true}) & (t_2, \mathsf{false}) & (t_3, \mathsf{false}) & (t_4, \mathsf{true}) & (t_5, \mathsf{true}) & (t_6, \mathsf{false}) \\ x \mapsto & & (t_2, \mathsf{true}) & & (t_4, \mathsf{true}) & & (t_6, \mathsf{true}) \end{array}$$

Notations We write $\mathcal{T}(s)$ for the chain of tags of a signal s and $\min s$ and $\max s$ for its minimal and maximal tag. We write $\mathcal{V}(b)$ for the domain of a behavior b (a set of signal names). The restriction of a behavior b to X is noted $b|_X$ (i.e. $\mathcal{V}(b|_X) = X$). Its complementary $b_{/X}$ satisfies $b = b|_X \uplus b_{/X}$ (i.e. $\mathcal{V}(b_{/X}) = \mathcal{V}(b) \setminus X$). We overload $\mathcal{T}$ and $\mathcal{V}$ to designate the tags of a behavior b and the set of signal names of a process p.

Since tags along a signal s form a chain $C = \mathcal{T}(s)$, we write C_i for the ith instant in chain C and have that $C_i \leq C_j$ iff $i \leq j$ for all $i, j \geq 0$.

Reaction A reaction r is a behavior with (at most) one time tag t. We write $\mathcal{T}(r)$ for the tag of a non empty reaction r. The empty signal is noted $\emptyset$. If a reaction r is concatenable to a behavior b, written $b \cdot^? r$ then the concatenation of r to b is written $b \cdot r$.

$$b \cdot^? c \Leftrightarrow \mathcal{V}(b) = \mathcal{V}(c) \wedge \max(\mathcal{T}(b)) < \min(\mathcal{T}(c)) \qquad b \cdot^? c \Rightarrow \forall x \in \mathcal{V}(b), (b \cdot c)(x) = b(x) \uplus r(x)$$

Synchronous structure A behavior c is a *stretching* of a behavior b, written $b \leq c$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and there exists a bijection f on tags s.t.

$\forall t, u, t \leq f(t) \wedge (t < u \Leftrightarrow f(t) < f(u))$

$\forall x \in \mathcal{V}(b), \mathcal{T}(c(x)) = f(\mathcal{T}(b(x))) \wedge \forall t \in \mathcal{T}(b(x)), b(x)(t) = c(x)(f(t))$

b and c are *clock-equivalent*, written $b \sim c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$. The synchronous composition $p \,|\, q$ of two processes p and q is defined by combining behaviors $b \in p$ and $c \in q$ that are identical on the interface between p and q : $I = \mathcal{V}(p) \cap \mathcal{V}(q)$.

$$p \,|\, q = \{b \cup c \,|\, (b, c) \in p \times q \ \wedge \ b|_I = c|_I \ \wedge \ I = \mathcal{V}(p) \cap \mathcal{V}(q)\}$$

Alternatively, the synchronous structure of a process can be interpreted as an equivalence relation $\sim$ that refines the causal tag structure of individual signals (for all $b \in \mathcal{B}$, for all $x \in \mathcal{V}(b)$, $\mathcal{T}(b(x))$ is a chain of $\mathbb{T}$). If we make the assumption that the only phenomenological structure is this causal structure, then the synchronous structure of a behavior $\sim^b$ can be seen as a way to slice time across individual signals in a way that preserves causality: for all $t, u \in \mathcal{T}(b)$, 1. $t < u$ iff $t \not\sim^b u$ and 2. $t \leq u$ iff $t \sim u$ or $t < v$ and $v \sim^b u$.

2.2 Formal semantics

We examplify the use of the model of computation by considering the kernel operators of Signal. The meaning $[\![P]\!]$ of a Signal process P is a set of behaviors that are inductively defined by the concatenation of reactions (we assume that $\emptyset_{\mathcal{V}(P)} \in [\![P]\!]$ for all P).

The meaning of the synchronous composition $P \,|\, Q$ is the synchronous composition $[\![P \,|\, Q]\!] = [\![P]\!] \,|\, [\![Q]\!]$ of the meaning of P and Q. The meaning of restriction is defined by $[\![P/x]\!] = \{c \,|\, b \in [\![P]\!] \wedge c \leq (b_{/x})\}$.

The semantics of a delay $x = y \,\mathsf{pre}\, v$ is defined by appending a reaction r of tag t to a behavior b. It initially defines x by the value v (when b is empty) and then by the previous value of y (i.e. $b(y)(u)$ where u is the maximal tag of b).

$$[\![x = y \,\mathsf{pre}\, v]\!] = \left\{ b \in \mathcal{B}|_{x,y} \;\middle|\; \begin{array}{l} \mathcal{T}(b(x)) = \mathcal{T}(b(y)) \\ \forall t \in \mathcal{T}(b(x)),\ b(x)(\mathrm{succ}_C(t)) = b(y)(t) \\ \qquad\qquad\qquad\quad b(x)(\min(C)) = v \end{array} \right\}$$

$$[\![x = y \, f \, z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \mathcal{T}(b(x)) = \mathcal{T}(b(y)) = \mathcal{T}(b(z)) \\ \forall t \in \mathcal{T}(b(x)),\ b(x)(t) = f(b(y)(t), b(z)(t)) \end{array} \right\}$$

Similarly, the semantics of a sampling $x = y \,\mathsf{when}\, z$ defines x by y when z is true. Finally, $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise. We write $\mathcal{T}_{\mathsf{true}}(s)\{t \in \mathcal{T}(s) \,|\, s(t) = \mathsf{true}\}$ for the chain of tags at which a signal s is true.

$$[\![x = y \,\mathsf{when}\, z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \mathcal{T}(b(x)) = \mathcal{T}_{\mathsf{true}}\,(b(y)) \cap \mathcal{T}(b(z)) \\ \forall t \in \mathcal{T}(b(x)),\ b(x)(t) = b(y)(t) \end{array} \right\}$$

$$[\![x = y \,\mathsf{default}\, z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \mathcal{T}(b(x)) = \mathcal{T}(b(y)) \cup \mathcal{T}(b(z))\} \\ \forall t \in \mathcal{T}(b(x)),\ b(x)(t) = \begin{cases} b(y)(t),\ t \in \mathcal{T}(b(y)) \\ b(z)(t),\ t \notin \mathcal{T}(b(y)) \end{cases} \end{array} \right\}$$

3 Interpretation and semantics of Synoptic

We formalize the semantics Synoptic by isolating the core syntactic categories that characterize its expressive capability: blocks, dataflows, actions and automata.

Blocks are the main structuring element of Synoptic. A block $\mathsf{block}\ x\ A$ defines a functional unit of compilation and of execution that can be called from many contexts and with different modes in the system under design. A block x encapsulates a functionality A that may consists of subblocks, automata and dataflows. A block x is implicity associated with two signals $x.\mathsf{trigger}$ and $x.\mathsf{reset}$. The signal $x.\mathsf{trigger}$ starts the execution of A. The specification A may then operate at its own pace until the next $x.\mathsf{trigger}$ is signaled. The signal $x.\mathsf{reset}$ is delivered to x at at some $x.\mathsf{trigger}$ and forces A to reset its state and variables to initial values.

$$(\mathit{blocks}) \qquad \mathsf{block}\ x\ A\ \mathit{where}\ A, B ::= \mathsf{block}\ x\ A \mid \mathsf{dataflow}\ x\ A \mid \mathsf{automaton}\ x\ A \mid A \,|\, B$$

Dataflows ensure the inter-connection between data (inputs and outputs) and events (e.g. trigger and reset signals) within a block. A flow can be the connection of an event x to an event y, written $\mathsf{event}\ x \to y$, data y and z combined by a simple operation f to form the flow x, written $\mathsf{data}\ y\ f\ z \to x$ or a delayed connection from y to x to implement feedback, written $\mathsf{data}\ y\ \mathsf{pre}\ v \to x$. The signal x is initially defined by $x_0 = v$. Then, at each occurrence $n > 0$ of the signal y, it takes its previous value $x_n = y_{n-1}$.

$$(\mathit{dataflow}) \qquad \mathsf{dataflow}\ x\ A\ \mathit{where}\ A, B ::= \mathsf{data}\ y\ \mathsf{pre}\ v \to x \mid \mathsf{data}\ y\ f\ z \to x \mid \mathsf{event}\ x \to y \mid A \,|\, B$$

The execution of a data-flow is controlled by its parent clock. A data-flow simultaneously executes each connection it is composed of every time it is triggered by its parent block.

Actions are sequences of operations on variables that are performed during the execution of automata. An assignment $x = y\ f\ z$ defines the new value of the variable x from the current values of y and z by the function f. The skip statement stores the new values of variables that have been defined before it, so that they become current past it. The conditional $\mathsf{if}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$ executes A if the current value of x is true and executes B otherwise. A sequence $A; B$ executes A and then B.

$$(\mathit{action}) \qquad A, B ::= \mathsf{skip} \mid x = y\ f\ z \mid \mathsf{if}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B \mid A; B$$

Automata schedule the execution of operations and blocks by performing timely guarded transitions. An automaton receives control from its trigger and reset signals $x\mathsf{trigger}$ and $x.\mathsf{reset}$ as specified by its parent block. When an automaton is first triggered, or when it is reset, its starts execution from its initial state, specified as $\mathsf{initial\ state}\ S$. On any state $S : \mathsf{do}\ A$, it performs the action A. From this state, it may perform an immediate transition to new state T, written $S \to^{\mathsf{on}\ x} T$, if the value of the current variable x is true. It may also perform a delayed transition to T, written $S \twoheadrightarrow^{\mathsf{on}\ x} T$, that waits until the next trigger to resume execution (in state T). If no transition condition applies, it then waits the next trigger and resumes execution in st ate S. States and transitions are composed as $A \,|\, B$.

$$(\mathit{automaton}) \qquad \mathsf{automaton}\ x\ A\ \mathit{where}\ A, B ::= \mathsf{state}\ S : \mathsf{do}\ A \mid S \to^{\mathsf{on}\ x} T \mid S \twoheadrightarrow^{\mathsf{on}\ x} T \mid A \,|\, B$$

The syntax for automata supports additional constructs to facilitate the definition of more complex policies. The initial (and terminal) state(s) of an automaton are explicitly specified by $\mathsf{initial\ state}\, S : \mathsf{do}\, A$ and $\mathsf{terminal\ state}\, S : \mathsf{do}\, A$ while, in the semantics, the initial state is just an initial or reset value of the state and a terminal state a state without outgoing transition. The actions performed in a state can be further decomposing into the actions of entering a state $\mathsf{state}\, S\, \mathsf{on\ entry\ do}\, A$, the action repetitively performed in a given state $\mathsf{state}\, S\, \mathsf{during\ do}\, A$, the action performed when leaving a state $\mathsf{state}\, S\, \mathsf{on\ exit\ do}\, A$, the action performed while leaving a state $S \twoheadrightarrow^{\mathsf{on}\, x\, \mathsf{do}\, A} T$.

$$\begin{aligned} A, B ::= {} & \mathsf{initial\ state}\, S : \mathsf{do}\, A \mid \mathsf{terminal\ state}\, S : \mathsf{do}\, A \mid \mathsf{state}\, S\, \mathsf{on\ entry\ do}\, A \\ & \mid \mathsf{state}\, S\, \mathsf{during\ do}\, A \mid \mathsf{state}\, S\, \mathsf{on\ exit\ do}\, A \mid S \twoheadrightarrow^{\mathsf{on}\, x\, \mathsf{do}\, A} T \end{aligned}$$

The timed execution of an automaton combines the behavior of an action or a data-flow. The execution of a delayed transition or of a stutter is controlled by an occurrence of the parent trigger signal (as for a data-flow). The execution of an immediate transition is performed without waiting for a trigger or a reset (as for an action).

3.1 Blocks

A block x receives control from its trigger and reset signals $x.\mathsf{trigger}$ and $x.\mathsf{trigger}$. The translation of a block $\mathsf{block}\, x\, A$ is the same as that of A but it is parameterized by the pair of trigger and reset signals.

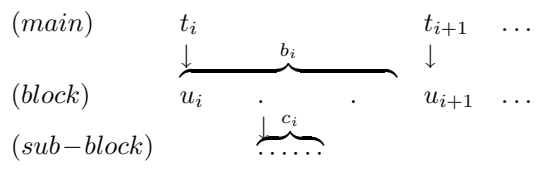
$$[\![\, \mathsf{block}\ x\, A]\!] = [\![A]\!]^{x.\mathsf{trigger}, x.\mathsf{reset}}$$

Denotation The corresponding denotation is parameterized by the time tags at which the trigger and reset events occur. They are defined by the chains C^t (for the trigger) and C^r (for the reset). Therefore, the operation of A is controlled by $C = (C^t, C^r)$.

$$[\![\, \mathsf{block}\ x\, A]\!] = \left\{ b \,|\, c \,|\, b \in [\![A]\!]^C,\ c \in \mathcal{B}|_{\substack{x.\mathsf{trigger} \\ x.\mathsf{reset}}},\ C^t = \mathcal{T}_{\mathsf{true}}\left(c(x.\mathsf{trigger})\right) \supseteq \mathcal{T}(c(x.\mathsf{reset})) = C^r \right\}$$

In addition to the above, an enable signal $x.\,\mathsf{enable}$ can be used to initiate the delayed execution of a block. Therefore, assuming $C^t = \mathcal{T}_{\mathsf{true}}\left(c(x.\mathsf{trigger})\right)$ and $C^e = \mathcal{T}_{\mathsf{true}}\left(c(x.\,\mathsf{enable}\,)\right)$, we should have that, $\forall t \in C^t \exists u \in C^e, t \leq u$.

Example The execution of a block is driven by the trigger t of its parent block. The block resynchronizes with that trigger every time, itself or one of its sub-block, performs an explicitly delayed transition, e.g. $S \twoheadrightarrow T$ for an automaton, or makes an explicit reference to time (e.g. skip for an action). Otherwise, the elapse of time shall not be sensed from within the block, whose operations (e.g., on c_i), should be perceived as belonging to the same period as within $[t_i, t_{i+1}[$. To implement this feature, we make use of an encoding of actions and automata using static single assignment. As a result, and from within a block, every immediate sequence of actions $A; B$ or transitions $A \to B$ only defines the value of the block's variable once, while defining several intermediate ones in the flow of its execution.



3.2 Dataflow

Data-flows are structurally similar to Signal programs and equally combined using synchronous composition. The interpretation $[\![A]\!]^{rt} = \langle P \rangle$ of a data-flow is parameterized by the reset and trigger signals of the parent block and returns a process P[1]. A delayed flow $\mathsf{data}\ y\,\mathsf{pre}\,v \to x$ initially defines x by the value v. It is reset to that value every time the reset signal r occurs. Otherwise, it takes the previous value of y in time. A functional flow $\mathsf{data}\ y\,f\,z \to x$ defines x by the product of (y, z) by f. An event flow $\mathsf{event}\ y \to x$ connects y to define x. The operator $?(y)$ converts an event y to a boolean data. The operator $\hat{}(y)$ converts the boolean data y to an event. We write $\mathrm{in}(A)$ and $\mathrm{out}(A)$ for the input and output signals of a dataflow A.

$$[\![\,\mathsf{dataflow}\ f\ A]\!]^{rt} = \langle [\![A]\!]^{rt} \,|\, \left(\prod_{x\in \mathrm{in}(A)} x\,\mathsf{sync}\,t\right)\rangle$$

$$[\![\,\mathsf{data}\ y\,\mathsf{pre}\,v \to x]\!]^{rt} = \langle x = (v\,\mathsf{when}\,r)\,\mathsf{default}\,(y\,\mathsf{pre}\,v) \,|\, (x\,\mathsf{sync}\,y)\rangle$$

$$[\![\,\mathsf{data}\ y\,f\,z \to x]\!]^{rt} = \langle x = y\,f\,z\rangle$$

$$[\![\,\mathsf{event}\ y \to x]\!]^{rt} = \langle x = \mathsf{when}\,y\rangle$$

$$[\![\,\mathsf{data}\ ?(y) \to x]\!]^{rt} = \langle x = \mathsf{true}\,\mathsf{when}\,y\,\mathsf{default}\,\mathsf{false} \,|\, x\,\mathsf{sync}\,t\rangle$$

$$[\![\,\mathsf{event}\ \hat{}(y) \to x]\!]^{rt} = \langle x = \mathsf{when}\,y\rangle$$

$$[\![A \,|\, B]\!]^{rt} = \langle [\![A]\!]^{rt} \,|\, [\![B]\!]^{rt}\rangle$$

By default, the convention of Synoptic is to synchronize the input signals of a data-flow to the parent trigger. It is however, possible to define alternative policies. One is to down-sample the input signals at the pace of the trigger. Another is to adapt or resample them at that trigger. Alternatively, adaptors could better be installed around the input and output signals of a block in order to resample them with respect to the specified frequency of the block.

$$[\![\mathsf{sample\ data}\ y \to x]\!]^{rt} = \langle x = y\,\mathsf{when}\,t\rangle \qquad [\![\mathsf{adapt\ data}\ y \to x]\!]^{rt} = \langle x = y\,\mathsf{cell}\,t\rangle$$

Denotation A data-flow $\mathsf{dataflow}\ f\ A$ is defined by the meaning of A controlled by the parent timing C. A delayed flow $\mathsf{data}\ y\,f\,z \to x$ assigns v to the signal x upon reset $t \in C^r$ and the previous value of y otherwise, at time $\mathrm{pred}_{C^x}(t)$. An operation $\mathsf{data}\ y\,f\,z \to x$ assigns the product of the values u and v of y and z by the operation f to the signal x. An event $\mathsf{event}\ x \to y$ triggers x every time y occurs. Composition $A \,|\, B$ merges all timely compatible traces of A and B under the same context.

$$[\![\,\mathsf{dataflow}\ f\ A]\!]^C = \{b \in [\![A]\!]^C \,|\, \forall x \in \mathrm{in}(A)\, C^t = \mathcal{T}(b(x))\}$$

$$[\![\,\mathsf{data}\ y\,\mathsf{pre}\,v \to x]\!]^C = \left\{ b \in \mathcal{B}|_{x,y} \;\middle|\; \begin{array}{l} C^x = \mathcal{T}(b(x)) = \mathcal{T}(b(y)) \supseteq C^v = C^r \cup \{\min(\mathcal{T}(b(x)))\} \\ \forall t \in C^x, b(x)(t) = \mathit{if}\ t \in C^v\ \mathit{then}\ v\ \mathit{else}\ b(y)(\mathrm{pred}_{C^x}(t)) \end{array} \right\}$$

$$[\![\,\mathsf{data}\ y\,f\,z \to x]\!]^C = \left\{ b \in \mathcal{B}|_{x,y,z} \,\middle|\, \forall t \in \mathcal{T}(b(x)) = \mathcal{T}(b(y)) = \mathcal{T}(b(z)),\ b(x)(t) = f(b(y)(t), b(z)(t)) \right\}$$

$$[\![\,\mathsf{event}\ y \to x]\!]^C = \{b \in \mathcal{B}|_{x,y} \quad | \mathcal{T}(b(x)) = \mathcal{T}(b(y))\,\}$$

$$[\![A \,|\, B]\!]^C = [\![A]\!]^C \,|\, [\![B]\!]^C$$

[1]The input term A and the output term P are marked by $[\![A]\!]$ and $\langle P\rangle$ for convenience

3.3 Actions

The execution of an action A starts at an occurrence of its parent trigger and shall end before the next occurrence of that event. During the execution of an action, one may also wait and synchronize with this event by issuing a skip statement.

A skip statement has no behavior but to signal the end of an instant: all the newly computed values of signals are flushed in memory and execution is resumed upon the next parent trigger. A statement $x!$ sends the signal x to its environment. Execution may continue within the same symbolic instant unless a second emission is performed: one shall issue a skip statement before that. An operation $x = y\ f\ z$ takes the current value of y and z to define the new value of x by the product with f. A conditional $\mathsf{if}\, x\, \mathsf{then}\, A\, \mathsf{else}\, B$ executes A or B depending on the current value of x.

As a result, only one new value of a variable x should at most be defined within an instant delimited by the start and the end or skip statement. Therefore, the interpretation of an action consists of its decomposition in static single assignment form. To this end, we use an environment E to associate each variable with its definition, an expression, and a guard, that locates it (in time).

An action holds an internal state s that stores an integer n denoting the current portion of the actions that is being executed. State 0 represents the start of the program and each $n > 0$ labels a skip statement that materializes a synchronized sequence of actions.

The interpretation $[\![A]\!]^{s,m,g,E} = \langle P \rangle_{n,h,F}$ of an action A takes as parameters the state variable s, the state m of the current section, the guard g that leads to it, and the environment E. It returns a process P, the state n and guard h past it, and an updated environment F. We write $\mathrm{use}^g_E(x)$ for the expression that returns the definition of the variable x at the guard g and $\mathrm{def}^g_E(x)$ for storing the final values of all variables x defined in E at the guard g.

$$\mathrm{use}^g_E(x) = if\ x \in \mathcal{V}(E)\ then\ \langle E(x) \rangle\ else\ \langle (x\ \mathsf{pre}\ 0)\ \mathsf{when}\ g \rangle \qquad \mathrm{def}_g(E) = \prod_{x \in \mathcal{V}(E)} (x = \mathrm{use}^g_E(x))$$

Execution is started with $s = 0$ upon receipt of a trigger t. It is also resumed from a skip statement at $s = n$ with a trigger t. Hence the signal t is synchronized to the state s of the action. The signal r is used to inform the parent block (an automaton) that the execution of the action has finished (it is back to its initial state 0). An end statement resets s to 0, stores all variables x defined in E with an equation $x = \mathrm{use}^g_E(x)$ and finally stops (its returned guard is false).

A skip statement advances s to the next label $n + 1$ when it receives control upon the guard e and flushes the variables defined so far. It returns a new guard $(s\ \mathsf{pre}\ 0) = n + 1$ to resume the actions past it. An action $x!$ emits x when its guard e is true. A sequence $A; B$ evaluates A to the process P and passes its state n_A, guard g_A, environment E_A to B. It returns $P \,|\, Q$ with the state, guard and environment of B.

$$\begin{aligned}
[\![\, \mathsf{do}\ A]\!]^{nt} &= \langle (P \,|\, s\ \mathsf{sync}\ t \,|\, r = (s = 0))\ /s \rangle \ \ and\ \langle P \rangle_{n,h,F} = [\![A;\ \mathsf{end}\,]\!]^{s,0,((s\ \mathsf{pre}\ 0)=0),\emptyset} \\
[\![\, \mathsf{end}\,]\!]^{s,n,g,E} &= \langle s = 0\ \mathsf{when}\ g \,|\, \mathrm{def}_g(E) \rangle_{0,\ \mathsf{false}\ ,\emptyset} \\
[\![\, \mathsf{skip}\,]\!]^{s,n,g,E} &= \langle s = n + 1\ \mathsf{when}\ g \,|\, \mathrm{def}_g(E) \rangle_{n+1,((s\ \mathsf{pre}\ 0)=n+1),\emptyset} \\
[\![x!]\!]^{s,n,g,E} &= \langle x =\ \mathsf{true}\ \mathsf{when}\ g \rangle_{n,g,E} \\
[\![x = y\ f\ z]\!]^{s,n,g,E} &= \langle x = e \rangle_{n,g,E_x \uplus \{x \mapsto e\}}\ where\ e = \langle f(\mathrm{use}^g_E(y), \mathrm{use}^g_E(z))\ \mathsf{when}\ g \rangle \\
[\![A; B]\!]^{s,n,g,E} &= \langle P \,|\, Q \rangle_{n_B,g_B,E_B}\ \ where\ \langle P \rangle_{n_A,g_A,E_A} = [\![A]\!]^{s,n,g,E} and\ \langle Q \rangle_{n_B,g_B,E_B} = [\![B]\!]^{s,n_A,g_A,E_A}
\end{aligned}$$

Similarly, a conditional evaluates A with the guard $g\ \mathsf{when}\ x$ to P and B with $g\ \mathsf{when\ not}\ x$ to Q. It returns $P \,|\, Q$ but with the guard $g_A\ \mathsf{default}\ g_B$. All variables $x \in X$, defined in both E_A and E_B,

are merged in the environment F.

$$[\![\mathsf{if}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]^{s,n,g,E} = \langle P \,|\, Q \rangle_{n_B,(g_A\ \mathsf{default}\ g_B),(E_A \bowtie E_B)}$$
$$where\ \langle P \rangle_{n_A,g_A,E_A} = [\![A]\!]^{s,n,(g\ \mathsf{when}\ \mathsf{use}^g_E(x)),E}$$
$$and\ \langle Q \rangle_{n_B,g_B,E_B} = [\![B]\!]^{s,n_A,(g\ \mathsf{when\ not}\ \mathsf{use}^g_E(x)),E}$$

We write $E \bowtie F$ to merge the definitions in the environments E and F.

$$\forall x \in \mathcal{V}(E) \cup \mathcal{V}(F),\ (E \bowtie F)(x) = \begin{cases} E(x), & x \in \mathcal{V}(E) \setminus \mathcal{V}(F) \\ F(x), & x \in \mathcal{V}(F) \setminus \mathcal{V}(E) \\ E(x)\ \mathsf{default}\ F(x), & x \in \mathcal{V}(E) \cap \mathcal{V}(F) \end{cases}$$

Notes An action cannot be reset from the parent clock because it is not synchronized to it. A sequence of emissions $x!; x!$ yield only one event along the signal x because they occur at the same (logical) time, as opposed to $x!;\ \mathsf{skip}\,; x!$ which sends the second one during the next trigger.

Denotation Given its state b, the execution $c_k = [\![A]\!]_b$ of an action A returns a new state c and a status k whose value is 1 if a skip occurred and 0 otherwise. We write $b_{\downarrow x} = b(x)(\min \mathcal{T}(b))$ and $b_{\uparrow x} = b(x)(\max \mathcal{T}(b))$ for the first and last value of x in b.

A sequence first evaluates A to c_k and then evaluates B with store $b \cdot c$ to d_l. If k is true, then a skip has occurred in A, meaning that c and d belong to different instants. In this case, the concatenation of b and c is returned.

If k is false, then the execution that ended in A continues in B at the same instant. Hence, variables defined in the last reaction of c must be merged to variables defined in the first reaction of d. To this end, we write $(b \bowtie c)(x) = b(x) \uplus c(x)$ for all $x \in \mathcal{V}(b)$ if $t = \max(\mathcal{T}(b)) = \min(\mathcal{T}(c))$.

$$[\![\mathsf{do}\ A]\!]_b = b \cdot c\, where\, c_k = [\![A]\!]_b$$
$$[\![\mathsf{skip}]\!]_b = \emptyset_1$$
$$[\![x!]\!]_b = \{(x,t,1)\}_0$$
$$[\![x = y\ f\ z]\!]_b = \{((x,t,f(b_{\uparrow y}, b_{\uparrow z}))\}_0$$
$$[\![\mathsf{if}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_b = if\ b_{\uparrow x}\ then\ \langle A \rangle_b\ else\ \langle B \rangle_b$$
$$[\![A; B]\!]_b = if\ k\ then\ (c \cdot d)_l\ else\ (c \bowtie d)_l\ where\ c_k = [\![A]\!]_b\ and\ d_l = [\![B]\!]_{b \cdot c}$$

Example Consider the simple sequential program of the introduction. Its static single assignment form is depicted in Fig. 3.

$$\begin{array}{l} x = 0; \\ \mathsf{if}\ y\ \ \mathsf{then}\ \{x = x + 1\} \\ \qquad \mathsf{else}\ \{x = x - 1;\ \mathsf{skip}\,;\ x = x - 1\} \\ \mathsf{end} \end{array}$$

As in GCC, it uses a ϕ-node, line 9 to merge the possible values x_2 and x_4 of x flowing from each branch of the if. Our interpretation implements this ϕ by a $\mathsf{default}$ equation that merges these two values with the third, x_3, that is stored into x just before the skip line 6. The interpretation of all assignment instructions in the program follows the same pattern[2]. Line 2, for instance, the value of

[2] In the actual translation, temporarly names $x_{1,\ldots 5}$ are substituted by the expression that defines them. We kept them in the figure for a matter of clarity.

x is x_1, which flows from line ^{1}. Its assignment to the new definition of x, namely x_2, is conditioned by the guard y on the path from line ^{1} to ^{2}. It is conditioned by the current state of the program, which needs to be 0, from line ^{1} to ^{6} and ^{9} (state 1 flows from line ^{7} to ^{9}, overlapping on the ϕ-node). Hence the equation $x_2 = x_1 + 1 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, y$.

Figure 3: Tracing the interpretation of a timed sequential program

$^0\, x_1 = 0;$	$x_1 = 0 \,\mathsf{when}\, (s=0)^1$
^{1} if y	$x_2 = x_1 + 1 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, y^2$
^{2} then $x_2 = x_1 + 1$	$x_3 = x_1 - 1 \,\mathsf{when}\, (s=0) \,\mathsf{when\ not}\, y^4$
^{3} else {	$x_4 = (x \,\mathsf{pre}\, 0) - 1 \,\mathsf{when}\, (s=1)^7$
^{4} $x_3 = x_1 - 1;$	$x = x_2 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, y^9$
^{5} $x = x_3;$	$\mathsf{default}\, x_3 \,\mathsf{when}\, (s=0) \,\mathsf{when\ not}\, y^4$
^{6} skip;	$\mathsf{default}\, x_4 \,\mathsf{when}\, (s=1)^9$
^{7} $x_4 = x - 1$	$s' = 0 \,\mathsf{when}\, (s=1)^8$
^{8} };	$\mathsf{default}\, 0 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, y^1$
^{9} $x = \phi(x_2, x_4)$	$\mathsf{default}\, 1 \,\mathsf{when}\, (s=0) \,\mathsf{when\ not}\, y^6$
end	$s = s' \,\mathsf{pre}\, 0$

3.4 Automata

An automaton describes a hierarchic structure consisting of actions execute upon entry in a state by immediate and delayed transitions. An immediate transition occurs during the period of time allocated to a trigger. Hence, it does not synchronize to it. Conversely, a delayed transition occurs upon synchronization with the next occurrence of the parent trigger event.

As a result, an automaton is partitioned in regions. Each region corresponds to the amount of calculation that can be performed within the period of a trigger, starting from a given initial state.

Notations We write $\rightarrow_A$ and $\twoheadrightarrow_A$ for the immediate and delayed transition relations of an automaton A. We write $\mathrm{pred}_{\rightarrow_A}(S)$ and $\mathrm{succ}_{\rightarrow_A}(S)$ (resp. $\mathrm{pred}_{\twoheadrightarrow_A}(S)$ and $\mathrm{succ}_{\twoheadrightarrow_A}(S)$) for the predecessor and successor states of the immediate (resp. delayed) transitions $\rightarrow_A$ (resp. $\twoheadrightarrow_A$) from a state S in an automaton A.

$$\mathrm{pred}_R(S) = \{T \mid (T, x, S) \in R\} \qquad \mathrm{succ}_R(S) = \{T \mid (S, x, T) \in R\}$$

We write $\vec{S}$ for the region of a state S. It is defined by an equivalence relation.

$$\forall S, T \in \mathcal{S}(A)\ ((S, x, T) \in \rightarrow_A) \Leftrightarrow \vec{S} = \vec{T}$$

For any state S of A, written $S \in \mathcal{S}(A)$, it is required that the restriction of $\rightarrow_A$ to the region $\vec{S}$ is acyclic. Notice that, still, a delayed transition may take place between two states of the same region.

Interpretation An automaton A is interpreted by a process $[\![\, \mathsf{automaton}\, x\, A]\!]^{rt}$ parameterized by its parent trigger and reset signals. The interpretation of A defines a local state s. It is synchronized to the parent trigger t. It is set to 0, the initial state, upon receipt of a reset signal r and, otherwise, takes the previous value of s', that denotes the next state. The interpretation of all states is performed concurrently.

We give all states S_i of an automaton A a unique integer label $i = \lceil S_i \rceil$ and designate with $\lceil A \rceil$ its number of states. S_0 is the initial state and, for each state of index i, we call A_i its action i and

x_{ij} the guard of an immediate or delayed transition from S_i to S_j.

$$[\![\,\mathsf{automaton}\,x\,A]\!]^{nt} = \langle \left(t\,\mathsf{sync}\,s \,|\, s = (0\,\mathsf{when}\,r)\,\mathsf{default}\,(s'\,\mathsf{pre}\,0) \,|\, \left(\prod_{S_i \in \mathcal{S}(A)} [\![S_i]\!]^s \right) \right) / ss' \rangle$$

The concurrent interpretation $[\![S_i]\!]^s$ of all states $0 \leq i < \lceil A \rceil$ is implemented by a series of recursive equations that define the meaning of each state S_i depending on the result obtained for its predecessors S_j in the same region. Since regions are acyclic, this system of equations has therefore a unique solution. The interpretation of state S_i starts with that of its actions A_i. An action A_i defines a local state s_i synchronized to the parent state $s = i$ of the automaton. The automaton stutters with $s' = s$ if the evaluation of the action is not finished: it is in a local state $s_i \neq 0$.

Interpreting the actions A_i requires the definition of a guard g_i and of an environment E_i. The guard g_i defines when A_i starts. It requires the local state to be 0 or the state S_i to receive control from a predecessor S_j in the same region (with the guard x_{ji}). The environment E_i is constructed by merging these F_j returned by its immediate predecessors S_j. Once these parameters are defined, the interpretation of A_i returns a process P_i together with an exit guard h_i and an environment F_i holding the value of all variables it defines.

Upon evaluation of A_i, delayed transition from S_i are checked. This is done by the definition of a process Q_i which, first, checks if the guard x_{ij} of a delayed transition from S_i evaluates to true with F_i. If so, variables defined in F_i are stored with $\mathrm{def}_{h_i}(F_i)$ and the state of the parent automaton becomes $s' = j$ at the guard h_i.

$$\begin{array}{lll}
\forall i < \lceil A \rceil & [\![S_i]\!]^s = \langle (P_i \,|\, Q_i \,|\, s_i\,\mathsf{sync}\,(s = i) \,|\, s' = s\,\mathsf{when}\,(s_i \neq 0))\,/s_i \rangle \\
\quad \textit{where} & \langle P_i \rangle_{n,h_i,F_i} = [\![A_i]\!]^{s_i,0,g_i,E_i} \\
\quad \textit{and} & E_i = \bowtie_{S_j \in \mathrm{pred}_{\rightarrow_A}(S_i)} F_j \\
\quad \textit{and} & g_i = \mathsf{true}\,\mathsf{when}\,((s_i\,\mathsf{pre}\,0) = 0)\,\mathsf{default}\,\left(\bigvee_{(S_j,x_{ji},S_i) \in \rightarrow_A} (\mathrm{use}_E(x_{ji})) \right) \\
\quad \textit{and} & Q_i = \prod_{(S_i,x_{ij},S_j) \in \twoheadrightarrow_A} \left(\mathrm{def}_{h_i\,\mathsf{when}\,(\mathrm{use}_{F_i}(x_{ij}))}(F_i) \,|\, s' = j\,\mathsf{when}\,h_i\,\mathsf{when}\,(\mathrm{use}_{F_i}(x_{ij})) \right)
\end{array}$$

We write $\prod_{i \leq n} P_i$ for a finite product of processes $\mathcal{P}_1 \,|\, \ldots P_n$, $\sum_{i \leq n} A_i$ for a finite sequence of actions $A_1; \ldots A_n$, $\bigvee_{i \leq n} e_i$ for a finite merge $e_1\,\mathsf{default}\,\ldots e_n$ and $\bigwedge_{i \leq n} e_i$ for a finite sampling $e_1\,\mathsf{when}\,\ldots e_n$.

Denotation An automaton x receives control from its trigger at the clock C^t and is reset at the clock C^r. Its meaning is hence parameterized by $C = (C^t, C^r)$. The meaning of its specifications $\langle A \rangle_b^s$ is parameterized by the finite trace b, that represents the store, by the variable s, that represents the current state of the automaton.

At the ith step of execution, given that it is in state j ($(b_i)_{\uparrow s} = j$) the automaton produces a finite behavior $b_{i+1} = \langle S_j \rangle_b^s$. This behavior must match the timeline of the trigger: $\mathcal{T}(b_i) \subseteq C^t$. It must to the initial state 0 is a reset occurs: $\forall t \in C^r$, $b_i(s)(t) = 0$.

$$[\![\,\mathsf{automaton}\;x\,A]\!]^C = \left\{ (\circ_{i \geq 0} b_i)\,/s \;\middle|\; \begin{array}{l} b_0 = \{(x, t_0, 0) \,|\, x \in \mathcal{V}(A) \cup \{s\}\} \\ \forall i \geq 0,\;\; b_{i+1} \in [\![S_j]\!]^s_{b_i} \\ \qquad\quad j = if\ \min(\mathcal{T}(b_{i+1})) \in C^r\ then\ 0\ else\ (b_i)_{\uparrow s} \\ \qquad\quad \mathcal{T}(b_{i+1}) \subseteq C^t \end{array} \right\}$$

When an automaton is in the state S_i, its action A_i is evaluated to $c \in \langle A_i \rangle_b$ given the store b. Then, immediate or delayed transitions departing from S_i are evaluated to return the final state d of the automaton.

$$[\![S_i]\!]^s_b = \left\{ \begin{array}{lll} [\![S_j]\!]^s_c, & (S_i, x_{ij}, S_j) \in \rightarrow_A \wedge\, c_{\uparrow x_{ij}} & \\ c \uplus \{(s,t,j)\}, & (S_i, x_{ij}, S_j) \in \twoheadrightarrow_A \wedge\, c_{\uparrow x_{ij}} & \textit{where}\quad c/s = [\![\,\mathsf{do}\,A_i]\!]_b \\ c \uplus \{(s,t,i)\}, & \textit{otherwise} & \textit{and}\quad t = \max \mathcal{T} c \end{array} \right.$$

Simplifications A generalized weak (or synchronized) transition from a state S to a state T translates into one leaving S_C (the exit substate of S) and entering T_S (the entry substate of T from S). In this translation, an automaton implicitly stutters the "during" action of its current state if no transition is applicable.

$$\left(\begin{array}{l} S: \\ \quad \mathsf{during\ do}\, A \\ \quad \mathsf{on\ exit\ do}\, B \end{array}\right) \twoheadrightarrow^{\mathsf{on}\ g\ \mathsf{do}\ C} \left(\begin{array}{l} T: \mathsf{on\ entry\ do}\, D \\ \\ \end{array}\right)$$

$$S: \mathsf{do}\, A \rightarrow^{\mathsf{on}\ g} S_T: \mathsf{do}\, B \quad \twoheadrightarrow \quad T_S: \mathsf{do}\, C; D \rightarrow T$$

Notes An oversampling automaton cannot be reset until it performs a transition synchronized to its trigger (a weak transition). A terminal state $S: \mathsf{terminal\ state}\, A$ is a state $S: \mathsf{do}\, A$ from which no transition exists. It can only be reset to the initial state.

Example Reconsider our previous sequential program

$$x = 0;\ \mathsf{if}\, y\, \mathsf{then}\, x = x + 1\, \mathsf{else}\, \{x = x - 1;\ \mathsf{skip}\,; x = x - 1\}\, \mathsf{end}$$

that is now represented by an automaton. The aim of our interpretation of automata is to merge, use and store the environments of variables defined in all the states $S_{1\ldots5}$ in order to provide its regions with an equivalent meaning as if it was defined by a sequential program. One can actually check that the translation of the automaton (right) is almost identical the that of the original program.

$$\begin{array}{l|l} S_0: \mathsf{do}\, x = 0 & |\, S_0 \rightarrow^{\mathsf{on}\ y} S_1 \\ & |\, S_0 \rightarrow^{\mathsf{on\ not}\ y} S_2 \\ S_1: \mathsf{do}\, x = x + 1 & |\, S_1 \rightarrow S_5 \\ S_2: \mathsf{do}\, x = x - 1 & |\, S_2 \twoheadrightarrow S_4 \\ S_4: \mathsf{do}\, x = x - 1 & |\, S_4 \rightarrow S_5 \\ S_5: \mathsf{end} & \end{array} \quad \left| \quad \begin{array}{l} s = s'\, \mathsf{pre}\, 0 \\ |\ x = 0 - 1\, \mathsf{when}\, (s = 0)\, \mathsf{when\ not}\, y \\ |\ s' = 1\, \mathsf{when}\, (s = 0)\, \mathsf{when\ not}\, y \\ |\ x = \quad 0 + 1\, \mathsf{when}\, (s = 0)\, \mathsf{when}\, y \\ \qquad \mathsf{default}\, (x\, \mathsf{pre}\, 0) - 1\, \mathsf{when}\, (s = 1) \\ |\ s' = 0\, \mathsf{when}\, (s = 0)\, \mathsf{when}\, y\, \mathsf{default}\, (s = 1) \end{array}\right.$$

3.5 Modularity

The above technique easily and compositionally generalizes to the modular programming framework under consideration in our project. It may indeed be suitable to have the capability of defining mode automata that operate blocks defined or compiled separately, and to allow block to possibly use shared variables.

In the present framework, this is done by associating each block with a def/use profile to register the association between the uses and definitions of global variables (e.g. x) with local variables (e.g. $x_{0,\ldots4}$). As an example, consider the following procedure f to increment the global integer variable x.

$$void\ f()\{x = x + 1\}$$

Our analysis locally records the definition and use of x with two integer variables x_0 and x_1. This forms the profile of f. Associating f with it amounts to declaring the association of x_0 with the use of x and of x_1 with the definition of x, as follows (or differently).

$$void\ f(int\ x_0\ \#use\ x, int\ *x_1\ \#def\ x)\{*x_1 = x_0 + 1\}$$

Next, had our mode automaton been defined by calling two external functions $f()$ and $g()$ to respectively increment and decrement x (Fig. 4, left), we would then just had to rewrite it in SSA form as follows (Fig. 4, right). The final x can itself be defined as a parameter to the hence SSA-encoded automaton, making the whole technique modular, compositional, hierarchical.

Figure 4: Modular interpretation of a mode automaton into data-flow equations

$$
\begin{array}{ll}
S_0: & \mathsf{do}\, x = 0 \\
& \mid S_0 \to^{\mathsf{on}\ y} S_1 \\
& \mid S_0 \to^{\mathsf{on\ not}\ y} S_2 \\
S_1: & \mathsf{do}\, f() \\
& \mid S_1 \to S_5 \\
S_2: & \mathsf{do}\, g() \\
& \mid S_2 \twoheadrightarrow S_4 \\
S_4: & \mathsf{do}\, g() \\
& \mid S_4 \to S_5 \\
S_5: & \mathsf{end}
\end{array}
\qquad
\begin{array}{ll}
S_0: & \mathsf{do}\, x_1 = 0 \\
& \mid S_0 \to^{\mathsf{on}\ y} S_1 \\
& \mid S_0 \to^{\mathsf{on\ not}\ y} S_2 \\
S_1: & \mathsf{do}\, f(x_1, \&x_2) \\
& \mid S_1 \to S_5 \\
S_2: & \mathsf{do}\, g(x_1, \&x_3) \\
& \mid S_2 \twoheadrightarrow S_4 \\
S_4: & \mathsf{do}\, g(x_3, \&x_4) \\
& \mid S_4 \to S_5 \\
S_5: & x = \phi(x_2, x_4); \mathsf{end}
\end{array}
$$

3.6 Periodic behaviors

A periodic structure can be defined by the refinement of a causal and synchronous structure. A signal s samples a signal r with period π and phase ϕ, written $s = \pi.r + \phi$ iff $(C, D) = \mathcal{T}(r, s)$ and, for all $i \geq 0$, $C_{\pi.i+\phi} = D_i$.

For any signal s, we can interpret the periodic structure of the chain $\mathcal{T}(s) = (t_n)_{n\geq 0}$ by a morphism Π to the series of intervals $([\pi \times n, \pi \times n + \phi[)_{n\geq 0}$.

A morphism Π is consistent with the synchronous structure $\sim^b$ of a behavior b iff, for all $t, u \in \mathcal{T}(b)$, 1. $t < u$ iff $\max \Pi(t) < \min \Pi(u)$ and 2. $t \sim^b u$ iff $\min \Pi(t) < \max \Pi(u)$ or $\min \Pi(u) < \max \Pi(t)$.

$$
\begin{aligned}
[\![\mathsf{block}\ B\ P\ \langle F \rangle]\!] &= [\![\mathsf{block}\ B\ P]\!]^C,\ C^t \in [\![F]\!] \\
[\![\mathsf{event}\ x \to y\ \langle F \rangle]\!] &= [\![\mathsf{event}\ x \to y]\!]^C,\ C^t \in [\![F]\!] \\
[\![\mathsf{frequency}\ n\ hz]\!] &= \{C \mid \forall i \geq 0, i/n \leq \Pi(C_i) < (i+1)/n\} \\
[\![m\ hz\ \mathsf{with\ burst}\ n\ \mathsf{every}\ p\ hz]\!] &= \left\{ C \left| \begin{array}{l} \forall i \geq 0,\ j = i\ div\ n,\ k = (i\ mod\ n) \\ j/p + k/m \leq \Pi(C_i) < j/p + (i+1\ mod\ n)/m \end{array} \right. \right\}
\end{aligned}
$$

4 Experimental Results

Case studies previously presented in [1] in the frame of the FOTOVP project [9] gave experimental evidences on the performance of our interpretation technique compared to more classical encoding techniques based on automata. Performance improve both for code generation (minimization of synchronization points) and for verification (minimization of intermediate states). The examples in [1] consist of concurrent imperative programs counting bits in parallel.

Fig. 6, in Appendix, depicts the intermediate generated code for the main AOCS module of Fig. 1. It comprises an interpretation of the automaton itself as well as its data-flow connections with the safe (SAFE) and nominal (NM) modes. The first six equations define the state transitions of the automaton. Then, a case statement defines how the block corresponding to each mode is called by the definition of its data-flow connections and timing constraints (tick, trigger, reset). The last fourteen equation define the connection that are within the lexical scope of the automaton. There are only two state variables (`_AOCS_aut_A_1_previousState` and `_AOCS_aut_A_1_zNextState`) as well as two explicitly shared variables (`_LO_CMD` and `_LO_ERR_ATT`). The process that interprets the AOCS automaton is compiled modularly and linked with its sub-modules `DataFlowModel_SAFE_dtf` and `DataFlowModel_NM_dtf`.

Figure 5: States and transitions for the SSA interpretation of parallel counters

program	vars	states	reachable	transitions	
2-bits parallel counters (property true)	24	2^{24}	36	116	0.15 s
2-bits parallel counters with variant (prop. false)	25	2^{25}	107	359	0.27 s
8-bits parallel counters (property true)	36	2^{36}	1.296	3.896	66 s
8-bits parallel counters with variant (prop. false)	37	2^{37}	328.715	1.117.223	124 s

5 Related Work

One common mis-conception about SSA is that since multiple assignments to the same variable are translated into assignments to multiple intermediate variables, the explosion of the number of variables introduces a huge overhead. As our encoding demonstrates,this is indeed not a problem as all of these intermediate variables are encoded into temporary signals and can even be substituted by the expression that defines them. They can also be dealt efficiently by model-checkers as they require no additional BDD node. Our experiments have shown that the number of *state* variables does not explode by SSA transformation but are rather reduced to the number of explicit time boundaries (a `skip`) in the source program.

Our approach and tools are based on previous studies and experimental results on the translation of imperative programs into synchronous equations using SSA transformation [7]. In related works, SSA-based modeling is primarily used for the purpose of efficiently compiling imperative programs with GCC [3] or LLVM [5], with emphasis on aggressive optimizations [4] generated code safety, runtime optimization and for symbolic verification [8].

Our technique uses the underlying model of computation of SME platform, dedicated to offering such capabilities (efficient code-generation and accelerated verification) for timed synchronous specifications such as in the high-level, domain-specific, programming environment Synoptic. One big advantage of our approach is that it creates a minimal number of transitions in the generated code: the portion of code between two explicit synchronizations is indeed encoded as a single reaction. By contrast, a naive approach would consists of translating each instruction with an explicit control-point and generate a huge automaton. Its encoding would then introduce state variables with a large number of possible values. Our SSA-based translation avoids this overhead.

6 Conclusions

We gave a thorough description of a technique that embeds heterogeneous programming and modeling concepts by interpreting them into an expressive, synchronous data-flow and multi-clocked model of computation. The development of this technique is based on the use of a static single assignment decomposition technique that is applied across the boundaries of the source diagrams yet in a well-typed and modular manner. It demonstrates a striking affinity between SSA and synchronous data-flow models of computation.

As a result, we obtain an efficient method for transforming a hierarchy of blocks consisting of actions (sequential Esterel-like programs), data-flow diagrams (to connect and time modules) and mode automata (to schedule or mode blocks) into a set of synchronous equations.

The impact of this new transformation technique is twofolds. With regards to code generation objectives, it minimizes needed resynchronizations between blocks in the system, potentially gaining substantial performances from way less communication. With regards to verification requirements, it minimizes the number of state variables across hierarchic automata and hence maximizes model checking capabilities.

References

[1] Besnard, L., Gautier, T, Moy, M., Talpin, J.-P., Johnson, K., Maraninchi, F. "Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form". Automated Verification of Critical Systems. EASST, 2009.

[2] Brunette, C., Talpin, J.-P., Gamatié, A., Gautier, T. "A metamodel for the design of polychronous systems"Journal of Logic and Algebraic Programming, Special Issue on Applying Concurrency Research to Industry. Elsevier, 2008.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". ACM Transactions on Programming Languages and Systems, 13(4): 451-490. ACM Press, October 1991.

[4] B. Hardekopf and C. Lin. "Semi-sparse flow-sensitive pointer analysis". Symposium on Principles of programming languages. ACM Press, 2009.

[5] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". International Symposium on Code Generation and Optimization. ACM Press, 2004.

[6] Le Guernic, P., Talpin, J.-P., Le Lann, J.-C. "Polychrony for system design". Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design. World Scientific, August 2003.

[7] Talpin, J.-P., Berner, D., Shukla, S., Le Guernic, P., Gamatié, A., Gupta, R. "Behavioral type inference for compositional system design". Chapter in Formal Methods and Models for System Design. Kluwer Academic Publishers, 2004.

[8] A. Zaks and R. Joshi. "Verifying multi-threaded C Programs with SPIN". International SPIN Workshop on Model Checking of Software. Springer, 2008.

[9] ANR project FotoVP `http://www-verimag.imag.fr/SYNCHRONE/fotovp`.

[10] ANR project Spacify `http://spacify.gforge.enseeiht.fr`.

[11] Polychrony and SME `http://www.irisa.fr/espresso/polychrony`.

Figure 6: Intermediate code for the main AOCS automaton

```
process Automaton_AOCS_aut =
  ( ? _POS_Data; _ATT_Data; _TO_NM; event _tick, _trigger, _reset; ! _CMD; _ERR_ATT; )
  (| subprocesses4 ::
    (| _WT_2_SAFE_TO_NM_P_0 := true when _TO_NM
         when (_AOCS_aut_A_1_currentState=#_SAFE)
     | _WT_3_NM_TO_SAFE_P_0 := true when _ERR_ATT
       when (_AOCS_aut_A_1_currentState=#_NM)
     | _AOCS_aut_A_1_currentState := _AOCS_aut_A_1_zNextState
     | _AOCS_aut_A_1_nextState := (#_NM when _WT_2_SAFE_TO_NM_P_0)
         default (#_SAFE when _WT_3_NM_TO_SAFE_P_0)
         default _AOCS_aut_A_1_currentState
     | _AOCS_aut_A_1_previousState := _AOCS_aut_A_1_currentState$ init #_SAFE
     | _AOCS_aut_A_1_zNextState := _AOCS_aut_A_1_nextState$ init #_SAFE
     | case _AOCS_aut_A_1_currentState in
     {#_SAFE}:
     (| subprocesses5 :: (| subprocesses6 ::
         (| _POS_Data1 := _SAFE_POS_Data
          | _ATT_Data1 := _SAFE_ATT_Data
          | _TO_NM2 := _SAFE_TO_NM
          | _tick2 := _SAFE_tick
          | _trigger2 := _SAFE_trigger
          | _reset2 := _SAFE_reset
          | _DataFlowModel_SAFE_dtf :: (_CMD1,_ERR_ATT2) :=
              DataFlowModel_SAFE_dtf{}
              (_POS_Data1,_ATT_Data1,_TO_NM2,_tick2,_trigger2,_reset2)
          | _SAFE_POS_Data := _POS_Data
          | _SAFE_ATT_Data := _ATT_Data
          | _SAFE_TO_NM := _TO_NM
          | _SAFE_tick := _tick
          | _SAFE_trigger := _clock_10_Hz
          | _SAFE_reset := _reset
          | _SAFE_CMD := _CMD1
          | _SAFE_ERR_ATT := _ERR_ATT2
          | _LO_CMD2 := _SAFE_CMD
          | _LO_ERR_ATT2 := _SAFE_ERR_ATT
          |)
          where _SAFE_POS_Data; _SAFE_ATT_Data; _SAFE_TO_NM;
            _SAFE_CMD; _SAFE_ERR_ATT; _CMD1; _ERR_ATT2;
            event _SAFE_tick, _SAFE_trigger, _SAFE_reset, _tick2, _trigger2, _reset2;
            label _DataFlowModel_SAFE_dtf, _POS_Data1, _ATT_Data1, _TO_NM2;
          end
        |) where label subprocesses6;
           end
      |)
     {#_NM}:
     (| subprocesses7 :: (| subprocesses8 ::
         (| _POS_Data2 := _NM_POS_Data
          | _ATT_Data2 := _NM_ATT_Data
          | _TO_NM3 := _NM_TO_NM
          | _tick3 := _NM_tick
          | _trigger3 := _NM_trigger
          | _reset3 := _NM_reset
          | _DataFlowModel_NM_dtf :: (_CMD2,_ERR_ATT3) :=
              DataFlowModel_NM_dtf{}
              (_POS_Data2,_ATT_Data2,_TO_NM3,_tick3,_trigger3,_reset3)
          | _NM_POS_Data := _POS_Data
          | _NM_ATT_Data := _ATT_Data
          | _NM_TO_NM := _TO_NM
          | _NM_tick := _tick
          | _NM_trigger := _clock_20_Hz
          | _NM_reset := _reset
          | _NM_CMD := _CMD2
          | _NM_ERR_ATT := _ERR_ATT3
          | _LO_CMD1 := _NM_CMD
          | _LO_ERR_ATT1 := _NM_ERR_ATT
          |) where _NM_POS_Data; _NM_ATT_Data; _NM_TO_NM; _NM_CMD;
               _NM_ERR_ATT; _CMD2; _ERR_ATT3;
               event _NM_tick, _NM_trigger, _NM_reset, _tick3,
               _trigger3, _reset3;
               label _DataFlowModel_NM_dtf, _POS_Data2,
               _ATT_Data2, _TO_NM3;
             end
        |) where label subprocesses8;
           end
      |)
    |) where type _AOCS_aut_A_1_type = enum (_SAFE, _NM);
         event _WT_2_SAFE_TO_NM_P_0, _WT_3_NM_TO_SAFE_P_0;
         _AOCS_aut_A_1_type _AOCS_aut_A_1_currentState,
         _AOCS_aut_A_1_nextState, _AOCS_aut_A_1_previousState,
         _AOCS_aut_A_1_zNextState;
         label subprocesses5, subprocesses7;
       end
  | (| _LO_CMD ::= _LO_CMD2
     | _LO_CMD ::= _LO_CMD1
     | _LO_ERR_ATT ::= _LO_ERR_ATT2
     | _LO_ERR_ATT ::= _LO_ERR_ATT1
     | _clock_10_Hz := _tick count 10
     | _XMI_Id_3 := (^_LO_ERR_ATT1) default (^_LO_ERR_ATT2)
     | _XMI_Id_4 := (^_LO_CMD1) default (^_LO_CMD2)
     | _clock_20_Hz := _tick count 5
     | _XMI_Id_4 ^= _LO_CMD
     | _XMI_Id_3 ^= _LO_ERR_ATT
     | _CMD := _LO_CMD
     | _ERR_ATT := _LO_ERR_ATT
     |)
  |) where shared _LO_CMD, event _LO_ERR_ATT;
       _clock_10_Hz; _clock_20_Hz; _XMI_Id_3; _XMI_Id_4; _LO_CMD1; _LO_CMD2;
       event _LO_ERR_ATT1, _LO_ERR_ATT2;
       label subprocesses4;
     end;
```

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399