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

Jean-Pierre Talpin, Julien Ouy, Thierry Gautier, Loïc Besnard
INRIA and CNRS
IRISA, Rennes, France
{Jean-Pierre.Talpin,Julien.Ouy,Thierry.Gautier,Loic.Besnard}@irisa.fr

Alexandre Cortier
CNES and IRIT
Université Paul Sabatier, Toulouse, France
cortier@irit.fr

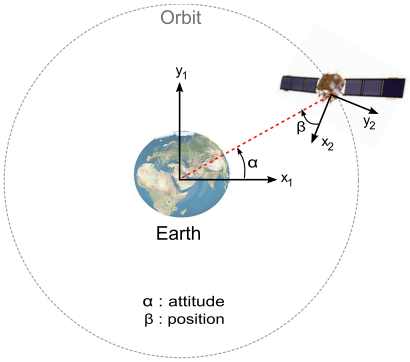
***Abstract*—The ANR project SPaCIFY develops a domain-specific programming environment, Synoptic, to engineer embedded software for space applications. Synoptic is an Eclipse-based modeling environment which supports all aspects of aerospace software design. As such, it is a domain-specific environment 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 significantly reduces the needed control states 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 reduces 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 also reduces the number of states across a hierarchy of automata and hence maximizes model checking performances.**

I. Introduction

A. Context

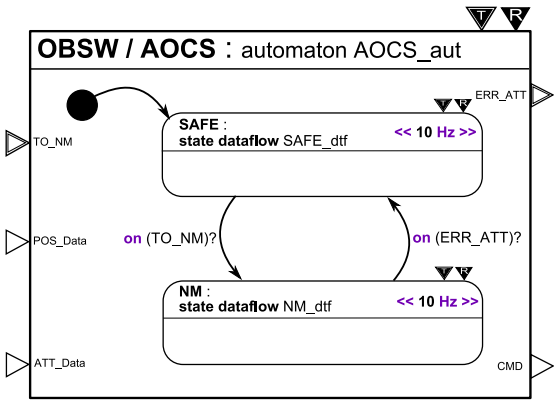
In the ANR project SPaCIFY [15], we develop a domain specific programming environment, Synoptic, dedicated to the design of space application and control software. Synoptic is an Eclipse-based modeling workbench based on Model Driven Engineering and formal methods which supports many aspects of aerospace 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 an Attitude and Orbit Control System (AOCS), Fig. 1.

Fig. 1. AOCS - Sattelite positionning software



This software system is responsible for automatically moving the satellite into a correct position (attitude and orbit) before initiating interaction with the ground. Its specification consists of the composition of a hierarchy of heterogeneous diagrams. Each diagram represents one specfic aspect of the software's role: automata, Fig. 2, or Fig. 5 on the next page, control its modes of operation for specific conditions (e.g. a solar eclipse).

Fig. 2. AOCS - Mode automaton of the sattelite positionning software



Data-flows, Fig. 3, define communication links and/or periodic processing tasks. Timed imperative programs, Fig. 4, specify sequential algorithmic behaviors.

B. 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 significantly reduces the needed control and variable states and block synchronizations. It consists of an inductive static-single assignment transformation algorithm across a hierarchy of blocks that produces synchronous equations.

C. 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 reduces the number and updates of states 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.

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

The static single assignment form of this program [5] 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.

$$
\begin{array}{l}
x_1 = 0; \\
\mathsf{if}\ y \\
\quad \mathsf{then}\ x_2 = x_1 + 1 \\
\quad \mathsf{else}\ \{x_3 = x_1 - 1; x = x_3;\ \mathsf{skip}\,;\, x_4 = x - 1\}; \\
x = \phi(x_2, x_4)
\end{array}
$$

It is interpreted 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. The signals s records the current state of the program (0 or 1).

$$
\begin{array}{l}
x_1 = 0\,\mathsf{when}\,(s = 0) \\
x_3 = x_1 - 1\,\mathsf{when}\,(s = 0)\,\mathsf{when\ not}\,y \\
x_4 = (x\,\mathsf{pre}\,0) - 1\,\mathsf{when}\,(s = 1) \\
x = x_2\,\mathsf{when}\,(s = 0)\,\mathsf{when}\,y\,\mathsf{default}\,x_4\,\mathsf{when}\,(s = 1) \\
\vdots
\end{array}
$$

In the present paper, we show how to not only translate the core imperative programming features into equations, 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.

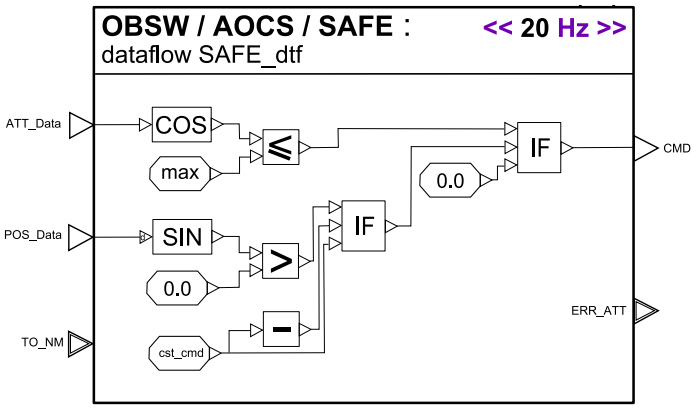
II. An overview of Synoptic

Blocks are the main structuring elements of Synoptic. A block 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 consist of sub-blocks, automata and dataflows. A block x is implicity associated with two signals x.trigger and x.reset. The signal x.trigger starts the execution of A. The specification A may then operate at its own pace until the next x.trigger is signaled. The signal x.reset is delivered to x at at some x.trigger and forces A to reset its state and variables to initial values.

$$
\begin{array}{lll}
(blocks) & A, B ::= & \mathsf{block}\ x\,A \\
& \mid & \mathsf{dataflow}\ x\,A \\
& \mid & \mathsf{automaton}\ x\,A \\
& \mid & A \,|\, B
\end{array}
$$

Data-flow diagrams: Data-flows ensure the interconnection between data (inputs and outputs) and events (e.g. trigger and reset signals) within a block. Fig. 3 gives a data-flow diagram of the AOCS software.

Fig. 3. AOCS - data-flow of the SAFE positionning mode



Three kinds of flows can be defined . An event flow connects an event x to an event y, written event $x \rightarrow y$. A data flow

$\mathsf{data}\ y\ f\ z \rightarrow x$ combines y and z by a simple operation f to form the flow x. A delayed flow $\mathsf{data}\ y\ \mathsf{pre}\ v \rightarrow x$ feeds y back to x: x is initially defined by $x_0 = v$ and then, at each occurrence $n > 0$ of the signal y, it takes its previous value $x_n = y_{n-1}$. 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.

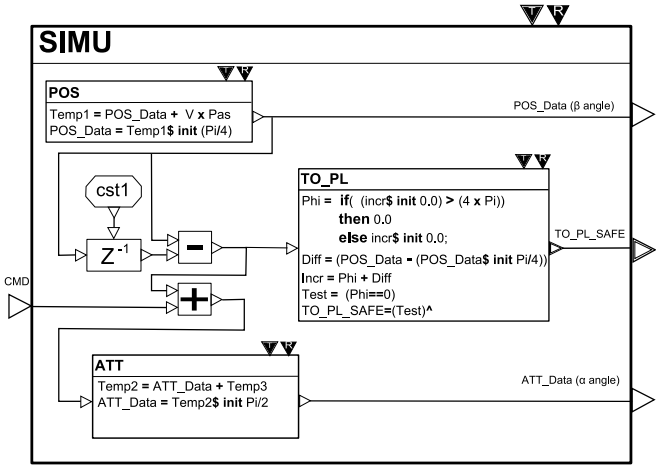
$$
\begin{array}{lll}
(dataflow) & A, B ::= & \mathsf{data}\ y\ \mathsf{pre}\ v \rightarrow x \\
& \mid & \mathsf{data}\ y\ f\ z \rightarrow x \\
& \mid & \mathsf{event}\ x \rightarrow y \\
& \mid & A \,|\, B
\end{array}
$$

Actions: Imperative programs define sequences of operations on variables that are performed during the execution of automata. Fig. 4 depicts action blocks in the simulation data-flow diagram of the AOCS software.

$$
\begin{array}{lll}
(action) & A, B ::= & \mathsf{skip} \\
& \mid & x = y\ f\ z \\
& \mid & x! \\
& \mid & \mathsf{if}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B \\
& \mid & A; B
\end{array}
$$

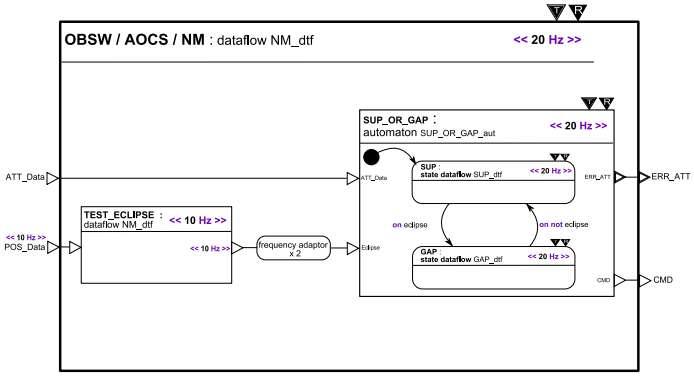
Assignment $x = y\ f\ z$ defines the new value of a variable x by applying f to the current values of y and z. The skip statement suspends execution until the next trigger event occurs. Upon this statement, new values of variables are stored and become current past the skip. Emission $x!$ triggers an event x. 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.

Fig. 4. AOCS - Action blocks in the data-flow diagram SIMU



Automata: Mode automata [10] schedule the execution of operations and blocks by performing timely guarded transitions. Fig. 5 gives the sub-automaton of the nominal mode in the AOCS software. An automaton receives control from its trigger and reset signals x.trigger and x.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 $S \rightarrow^{\mathsf{on}\ x} T$, if the value of the current variable x is true.

Fig. 5. AOCS - Mode automaton for the nominal (NM) positionning mode



It may also perform a delayed transition to T, written $S \twoheadrightarrow^{\mathsf{on}\ x} T$, that waits the next trigger before to resume execution (in state T).

$$
\begin{array}{lll}
(automaton) & A, B ::= & \mathsf{state}\ S : \mathsf{do}\ A \\
& \mid & S \rightarrow^{\mathsf{on}\ x} T \\
& \mid & S \twoheadrightarrow^{\mathsf{on}\ x} T \\
& \mid & A \,|\, B
\end{array}
$$

If no transition condition applies, it then waits the next trigger and resumes execution in st ate S. States and transitions are composed as $A \,|\, B$. 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).

III. Model of Computation

The model of computation, on which Synoptic relies for program transformation and code generation, is that of the Eclipse-based synchronous modeling environment SME [16]. The core of SME is based on the synchronous programming language Signal [9]. 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 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. A functional equation $x = y\ f\ z$ defines x by applying the boolean or arithmetic operator f to the values of 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.

Clocks: 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.

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

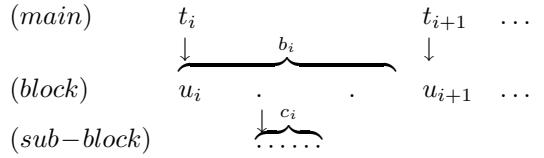
Notice that this equation does not define a Kahn process: the clock of the output signal counter is faster than that of the input signal reset. One may later refine it by adding a clock equation counter sync reset to generate a count every time an occurrence of the input signal reset, true or false, is present.

Fig. 6. Chronogram displaying an execution of the counter

counter	1	2	3	0	1	2	3	0
counter pre 0	0	1	2	3	0	1	2	3
reset				$\top$				$\top$
0 when reset				0				0

IV. Interpretation of Synoptic

In Synoptic, the execution of a block is driven by the trigger t of its parent block. The block resynchronizes with that trigger when it 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.



A. 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 (Fig. 7) is parameterized by the reset and trigger signals of the parent block and returns a process P (For the sake of clarity, $[\![A]\!]$ denotes the interpretation of the term A and brackets $\langle P \rangle$ delimit the result of the interpretation). 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.

In Fig. 7, we write $\prod_{i \le n} P_i$ for a finite product of processes $P_1 \,|\, \ldots P_n$. Similarly, $\bigvee_{i \le n} e_i$ a finite merge $e_1\, \mathsf{default}\, \ldots e_n$ and $\bigwedge_{i \le n} e_i$ a finite sampling $e_1\, \mathsf{when}\, \ldots e_n$.

Fig. 7. Interpretation of data-flow connections

$$[\![\, \mathsf{dataflow}\ f\ A]\!]^{rt} = \langle [\![A]\!]^{rt} \,|\, \left(\prod_{x \in \mathrm{in}(A)} x\, \mathsf{sync}\, t\right) \rangle$$
$$[\![\, \mathsf{data}\ y\, \mathsf{pre}\, v \to x]\!]^{rt} = \langle x = (v\, \mathsf{when}\, r)\, \mathsf{default}\, (y\, \mathsf{pre}\, v) \rangle$$
$$[\![\, \mathsf{data}\ y\, f\, z \to x]\!]^{rt} = \langle x = y\, f\, z \rangle$$
$$[\![\, \mathsf{event}\ y \to x]\!]^{rt} = \langle x = \ \mathsf{when}\, y \rangle$$
$$[\![A \,|\, B]\!]^{rt} = \langle [\![A]\!]^{rt} \,|\, [\![B]\!]^{rt} \rangle$$

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. Particular cases are the operator $?(y)$ to convert an event y to a boolean data and the operator $\hat{}(y)$ to convert 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.

Adaptors: By default, Synoptic synchronizes 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 using adaptors. The directive sample samples a faster signal y at the slower clock t of a block. The directive adapt over-samples a slower signal y at the faster clock t of a block. Its interpretation uses the Signal process cell that stores y in a register and loads its current value at the clock t.

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

B. 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.

$$\begin{array}{lll} (action) & A, B ::= & \mathsf{skip} \mid x! \\ & \mid & x = y\, f\, z \\ & \mid & \mathsf{if}\, x\, \mathsf{then}\, A\, \mathsf{else}\, B \\ & \mid & A; B \end{array}$$

A skip 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. Action $x!$ 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 before that. An operation $x = y\, f\, z$ takes the current value of y and z to define the new value of x by applying f to y and z. 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, at most one value of a variable x should be defined within an instant delimited by a start and an end or a skip. 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 that materializes a synchronized sequence of actions.

The interpretation $[\![A]\!]^{s,m,g,E} = \langle P\rangle_{n,h,F}$ of an action A has four parameters: a 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 of its continuation, and an updated environment F. The set of variables defined in E is written $\mathcal{V}(E)$. 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.

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

Execution is started with $s = 0$ upon receipt of a trigger t. It is also resumed from a skip 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).

$$[\![\,\mathsf{do}\, A]\!]^{rt} = \langle (P \,|\, s \,\mathsf{sync}\, t \,|\, r = (s = 0))\,/s\rangle$$
$$where \quad \langle P\rangle_{n,h,F} = [\![A;\, \mathsf{end}\,]\!]^{s,0,((s\,\mathsf{pre}\,0)=0),\emptyset}$$

An end 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 0).

$$[\![\,\mathsf{end}\,]\!]^{s,n,g,E} = \langle s = 0 \,\mathsf{when}\, g \,|\, \mathrm{def}_g(E)\rangle_{0,0,\emptyset}$$

A skip 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.

$$[\![\,\mathsf{skip}\,]\!]^{s,n,g,E} = \langle s = n+1 \,\mathsf{when}\, g \,|\, \mathrm{def}_g(E)\rangle_{\substack{n+1,\\ (s\,\mathsf{pre}\,0)=n+1,0}}$$

An action $x!$ emits x when the guard g is true. An operation $x = y\ f\ z$ defines the "def" of x with the "use" of y and z at the guard g.

$$[\![x!]\!]^{s,n,g,E} = \langle x = 1 \,\mathsf{when}\, g\rangle_{n,g,E}$$
$$[\![x = y\ f\ z]\!]^{s,n,g,E} = \langle x = e\rangle_{n,g,E_x \uplus \{x\mapsto e\}}$$
$$where \quad e = \langle f(\mathrm{use}^g_E(y), \mathrm{use}^g_E(z)) \,\mathsf{when}\, g\rangle$$

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.

$$[\![A; B]\!]^{s,n,g,E} = \langle P \,|\, Q\rangle_{n_B,g_B,E_B}$$
$$where \quad \langle P\rangle_{n_A,g_A,E_A} = [\![A]\!]^{s,n,g,E}$$
$$and \quad \langle Q\rangle_{n_B,g_B,E_B} = [\![B]\!]^{s,n_A,g_A,E_A}$$

Similarly, a conditional evaluates A with the guard $g \,\mathsf{when}\, x$ to P and B with $g \,\mathsf{when}\, \mathsf{not}\, x$ 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.

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

We write $E \uplus F$ to merge the definitions in the environments E and F. For all variables $x \in \mathcal{V}(E) \cup \mathcal{V}(F)$ in the domains of E and F,

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

Note that 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.

Example: Consider the simple sequential program of the introduction. Its static single assignment form is depicted in Fig. 8.

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

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 default equation that merges these two values with the third, x_3, that is stored into x just before the skip line 6.

Fig. 8. Tracing the interpretation of a timed sequential program

0. $x_1 = 0$;	$x_1 = 0 \,\mathsf{when}\, (s=0)^{(0)}$
1. if y	$x_2 = x_1 + 1 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, y^{(2)}$
2. then $x_2 = x_1 + 1$	$x_3 = x_1 - 1 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, \neg y^{(4)}$
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. $x = x_3$;	default $x_3 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, \neg y^{(5)}$
6. skip ;	default $x_4 \,\mathsf{when}\, (s=1)^{(9)}$
7. $x_4 = x - 1$	$s' = 0 \,\mathsf{when}\, (s=1)^{(8)}$
8. };	default $0 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, y^{(1)}$
9. $x = \phi(x_2, x_4)$	default $1 \,\mathsf{when}\, (s=0) \,\mathsf{when}\, \neg y^{(6)}$
end	$s = s' \,\mathsf{pre}\, 0$

The interpretation of all assignment instructions in the program follows the same pattern (in the actual translation, temporary names $x_{1,\ldots5}$ are substituted by the expression that defines them. We kept them in the figure for a matter of clarity). Line 2, for instance, the value of 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 $x_2 = x_1 + 1 \,\mathsf{when}\, (s{=}0) \,\mathsf{when}\, y$.

C. Automata

An automaton describes a hierarchic structure consisting of actions that are executed 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

$$\begin{aligned} &\mathrm{pred}_{\rightarrow_A}(S) = \{T \mid (T,x,S) \in \rightarrow_A\} \\ \text{and}\quad &\mathrm{succ}_{\rightarrow_A}(S) = \{T \mid (S,x,T) \in \rightarrow_A\} \end{aligned}$$

and, 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.Finally, 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.

$$[\![\,\mathsf{automaton}\, x\, A]\!]^{rt} = \left(\begin{array}{l} t \,\mathsf{sync}\, s \\ \mid\ s = (0 \,\mathsf{when}\, r) \,\mathsf{default}\, (s' \,\mathsf{pre}\, 0) \\ \mid\ \left(\prod_{S_i \in \mathcal{S}(A)} [\![S_i]\!]^s\right) \end{array}\right) / ss'$$

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.

The interpretation $[\![S_i]\!]^s$ of all states $0 \leq i < \lceil A \rceil$ of an automaton (Fig. 9) is implemented by a series of mutually 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 a region is by definition acyclic, this system of equations has therefore a unique solution.

$$[\![S_i]\!]^s = (P_i \,|\, Q_i \,|\, R_i \,|\, s_i \,\mathsf{sync}\, \mathsf{when}\, (s{=}i) \,|\, s' {=} s'_i)\,/s_i$$

Fig. 9. Recursive interpretation of a mode automaton

$$\begin{array}{l} \forall i < \lceil A \rceil,\ [\![S_i]\!]^s = (P_i \,|\, Q_i \,|\, R_i \,|\, s_i \,\mathsf{sync}\, \mathsf{when}\, (s{=}i) \,|\, s' {=} s'_i)\,/s_i \\ \textit{where} \\ \quad \langle P_i \rangle_{n,h_i,F_i} = [\![A_i]\!]^{s_i,0,g_i,E_i} \\ \quad E_i = \biguplus_{S_j \in \mathrm{pred}_{\rightarrow_A}(S_i)} F_j \\ \quad g_i = 1 \,\mathsf{when}\, (s_i \,\mathsf{pre}\, 0 = 0) \,\mathsf{default}\, h_i \\ \quad h_i = \bigvee_{(S_j,x_{ji},S_i) \in \rightarrow_A} (\mathrm{use}_E(x_{ji})) \\ \quad Q_i = \prod_{(S_i,x_{ij},S_j) \in \twoheadrightarrow_A} \left(\mathrm{def}_{h_i \,\mathsf{when}\, (\mathrm{use}_{F_i}(x_{ij}))}(F_i)\right) \\ \quad s'_i = (s \,\mathsf{when}\, s_i \neq 0) \,\mathsf{default}\, \left(\bigvee_{(S_i,x_{ij},S_j) \in \twoheadrightarrow_A} (j \,\mathsf{when}\, g_{ij})\right) \\ \quad g_{ij} = h_i \,\mathsf{when}\, (\mathrm{use}_{F_i}(x_{ij})),\ \forall (S_i,x_{ij},S_j) \in \twoheadrightarrow_A \end{array}$$

The interpretation of state S_i, Fig. 9, 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.

$$\langle P_i \rangle_{n,h_i,F_i} = [\![A_i]\!]^{s_i,0,g_i,E_i}$$

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}).

$$g_i = 1 \,\mathsf{when}\, (s_i \,\mathsf{pre}\, 0 = 0) \,\mathsf{default}\, \overbrace{\bigvee_{(S_j,x_{ji},S_i) \in \rightarrow_A} (\mathrm{use}_E(x_{ji}))}^{h_i}$$

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.

$$E_i = \biguplus_{S_j \in \mathrm{pred}_{\rightarrow_A}(S_i)} F_j$$

Upon termination of A_i, delayed transition from S_i are checked. This is performed by the 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 $\mathrm{def}_{h_i}(F_i)$.

$$Q_i = \prod_{(S_i,x_{ij},S_j) \in \twoheadrightarrow_A} \left(\mathrm{def}_{h_i \,\mathsf{when}\, (\mathrm{use}_{F_i}(x_{ij}))}(F_i)\right)$$

If the evaluation of action A_i is not finished, the automaton "stutters" by remaining in the local state s'_i of region i with $s' = s'_i$ and $s'_i = s \,\mathsf{when}\, (s_i \neq 0)$.

$$s' = s'_i \,|\, s'_i = (s \,\mathsf{when}\, s_i \neq 0)$$

Otherwise, a delayed transition from state i to state j, guarded by x_{ij}, may be fired. The boolean condition g_{ij} defines the value of the guard x_{ij}.

$$s'_i = \left(\bigvee_{(S_i,x_{ij},S_j) \in \twoheadrightarrow_A} \left(j \,\mathsf{when}\, \overbrace{h_i \,\mathsf{when}\, (\mathrm{use}_{F_i}(x_{ij}))}^{g_{ij}}\right)\right)$$

The next state equation $s' = s'_i$ produced on each state i is composed with the others to form the product $\prod_{i<\lceil A\rceil} s' = s'_i$ that is merged as $s' = \bigvee_{i<\lceil A\rceil} s'_i$.

Example: Let us reconsider our dummy sequential program, and now represent by a mode automaton (Fig. 10, left).

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

Our interpretation of automata merges, loads and stores the variables defined in states $S_{1\ldots5}$. Thanks to its decomposition in regions, we can associate it with an SSA interpretation of equivalent meaning (Fig. 10, right).

Fig. 10. SSA interpretation of a mode automaton into data-flow equations

$$\begin{array}{ll|ll} S_0: & \mathsf{do}\, x = 0 & S_0: & \mathsf{do}\, x_1 = 0 \\ & \mid S_0 \to^{\mathsf{on}\, y} S_1 & & \mid S_0 \to^{\mathsf{on}\, y} S_1 \\ & \mid S_0 \to^{\mathsf{on\ not}\, y} S_2 & & \mid S_0 \to^{\mathsf{on\ not}\, y} S_2 \\ S_1: & \mathsf{do}\, x = x + 1 & S_1: & \mathsf{do}\, x_2 = x_1 + 1 \\ & \mid S_1 \to S_5 & & \mid S_1 \to S_5 \\ S_2: & \mathsf{do}\, x = x - 1 & S_2: & \mathsf{do}\, x_3 = x_1 - 1; x = x_3 \\ & \mid S_2 \twoheadrightarrow S_4 & & \mid S_2 \twoheadrightarrow S_4 \\ S_4: & \mathsf{do}\, x = x - 1 & S_4: & \mathsf{do}\, x_4 = x - 1 \\ & \mid S_4 \to S_5 & & \mid S_4 \to S_5 \\ S_5: & \mathsf{end} & S_5: & x = \phi(x_2, x_4);\ \mathsf{end} \end{array}$$

One can actually check that the translation of the automaton is identical to that of the original program modulo substitution of local the local signals $x_{1,\ldots4}$ by their definition.

$$\begin{array}{l} x = 0 - 1\ \mathsf{when}\ (s = 0)\ \mathsf{when\ not}\, y \\ \qquad\qquad \mathsf{default}\, 0 + 1\ \mathsf{when}\ (s = 0)\ \mathsf{when}\, y \\ \qquad\qquad \mathsf{default}\ (x\ \mathsf{pre}\, 0) - 1\ \mathsf{when}\ (s = 1) \\ \mid s' = 1\ \mathsf{when}\ (s = 0)\ \mathsf{when\ not}\, y \\ \qquad\qquad \mathsf{default}\, 0\ \mathsf{when}\ (s = 0)\ \mathsf{when}\, y \\ \qquad\qquad \mathsf{default}\, 0\ \mathsf{when}\ (s = 1) \\ \mid s = s'\ \mathsf{pre}\, 0 \end{array}$$

V. Modularity

The above technique easily and compositionally generalizes to the modular programming framework under consideration in our project. This is simply 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 variable x.

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

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).

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

Had our mode automaton been defined by calling two external functions $f()$ and $g()$ to respectively increment and decrement x (Fig. 13, left), we would then just had to rewrite it in SSA form as follows (Fig. 13, right). The final x can itself be defined as a parameter to the hence SSA-encoded automaton, making the whole technique modular, compositional and hierarchical.

Fig. 11. Modular interpretation of a mode automaton into data-flow equations

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

VI. Experimental Results

Case studies previously presented in [1] in the frame of the FOTOVP project [14] gave experimental evidences on the performance of our interpretation technique compared to more classical encoding techniques based on automata, Fig. 12. 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. 12. 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

To convince oneself on the performance of our approach in giving a state-space efficient interpretation of imperative programs, Fig. 13 depicts a simple yet naive encoding (right) of our introductory program (left).

Fig. 13. Naive interpretation of a timed sequential program

1. $x = 0;$	$x' = 0\ \mathsf{when}\ (s = 1)$
2. $\mathsf{if}\ y$	$\quad \mathsf{default}\, x + 1\ \mathsf{when}\ (s = 3)$
$\quad \mathsf{then}$	$\quad \mathsf{default}\, x - 1\ \mathsf{when}\ (s = 4)$
3. $\quad x = x + 1$	$\quad \mathsf{default}\, x - 1\ \mathsf{when}\ (s = 6)$
$\quad \mathsf{else}\, \{$	$s' = 1\ \mathsf{when}\ (s = 6)$
4. $\quad x = x - 1;$	$\quad \mathsf{default}\, 2\ \mathsf{when}\ (s = 1)$
5. $\quad \mathsf{skip}\,;$	$\quad \mathsf{default}\, 3\ \mathsf{when}\ (s = 2)\ \mathsf{when}\, y$
6. $\quad x = x - 1$	$\quad \mathsf{default}\, 4\ \mathsf{when}\ (s = 2)\ \mathsf{when\ not}\, y$
$\quad \};$	$\quad \mathsf{default}\, 5\ \mathsf{when}\ (s = 4)$
end	$\quad \mathsf{default}\, 6\ \mathsf{when}\ (s = 5)$
	$s = s'\ \mathsf{pre}\, 1$
	$x = x'\ \mathsf{pre}\, 0$

This encoding associates each instruction of the original program with a state identified with a numerical label. This label is used to guard each transition in the control flow and and each operation in the data flow. Each branch in the control-flow is associated with a guarded assignment of the new state s' given the current state s and a possible condition (e.g. y). Each instruction in the data flow is associated with a guarded assignment of the new variable value x' using its current value x. Although it is simpler than our SSA-based interpretation, one easily observes that the present scheme generates significantly more states than in Fig. 8, more precisely, it associates a state with each instruction instead of one with each skip or each loop.

VII. 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, and that 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 approach and tools are based on previous studies and experimental results on the translation of imperative programs into synchronous equations using SSA transformation [12]. In related works, SSA-based modeling is primarily used for the purpose of efficiently compiling imperative programs with GCC [5] or LLVM [8], with emphasis on aggressive optimizations [7] generated code safety, runtime optimization and for symbolic verification [13].

Our technique uses the underlying model of computation of the 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 very few transitions in the generated model: one per explicit synchronization or control-loop. By contrast, the naive approach of Fig. 13 translates each instruction with an explicit control-point and ends up generating a huge automaton.

The present case study focuses on interpreting domain-specific modeling diagrams into a suitable model of computation for producing an executable GALS specification. The methodology we use, however, can directly be applied to the related and more general issue of co-modeling. Co-modeling is the joint interpretation of functional (e.g. Simulink) and structural (e.g. AADL) specifications. It aims at producing a formal specification suitable for architecture exploration: simulation, test, verification, and validation of a given embedded architecture for, e.g., predicting the effectiveness and performance of the system under design as early as possible.

On the other hand, co-modeling can be addressed by considering other models of computation than the one of Polychrony. Ptolemy [17], for instance, allows to model system described with heterogeneous formalisms and in different models of computation by using the notion of director to interface the components of a given model. In a similar manner, yet without using directors but, instead, a quasi-synchronous protocol [3], the Lustre data-flow language can be used to model some of the elements considered in this paper, to model, e.g. Simulink diagrams [4] or, e.g., AADl diagrams [6]. Our approach neither requires directors nor interface protocols.

Also, in programming languages such as SpecC [18] or SystemC [19], both the architecture and its functionalities are described (or constructuted) with the same C-like programming language. Maraninchi et al. [11] develop verification tools that model such hybrid (functional and structural) SystemC descriptions using automata and, based on joint preliminary results [1], plan to use an SSA intermediate form.

VIII. 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 significantly reduces needed resynchronizations between blocks in the system, potentially gaining substantial performances from way less communication. With regards to verification requirements, it also reduces 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] P. Caspi , C. Mazuet , N. Reynaud. "About the design of distributed control systems, the quasi-synchronous approach". International Conference on Computer Safety, Reliability and Security. Lecture Notes in Computer Science v. 2187. Springer, 2001.

[4] P. Caspi, A. Curic, A. Maignan, C. Sofronis and S. Tripakis. "Translating Discrete-Time Simulink to Lustre". In Transactions on Embedded Computing Systems, v. 4(4). ACM Press, 2005.

[5] 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.

[6] N. Halbwachs, E. Jahier, P. Raymond, X. Nicollin, D. Lesens. "Virtual execution of AADL models via a translation into synchronous programs". Embedded Software Conference. ACM Press, 2007.
[7] B. Hardekopf and C. Lin. "Semi-sparse flow-sensitive pointer analysis". Symposium on Principles of programming languages. ACM Press, 2009.
[8] 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.
[9] 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.
[10] F. Maraninchi and Y. Rémond. "Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems". Science of Computer Programming, v. 46. Elsevier, 2003.
[11] M. Moy, F. Maraninchi and L. Maillet-Contoz. "LusSy, a toolbox for the analysis of systems-on-a-chip at transactional level. Applications of Concurrency to System Design. IEEE Press, 2005.
[12] 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.
[13] A. Zaks and R. Joshi. "Verifying multi-threaded C Programs with SPIN". International SPIN Workshop on Model Checking of Software. Springer, 2008.
[14] Project FotoVP http://www-verimag.imag.fr/SYNCHRONE/fotovp.
[15] Project Spacify http://spacify.gforge.enseeiht.fr.
[16] Polychrony http://www.irisa.fr/espresso/polychrony.
[17] Ptolemy project http://ptolemy.eecs.berkeley.edu.
[18] SpecC System http://www.cecs.uci.edu/~specc.
[19] SystemC Initiative http://www.systemc.org.

APPENDIX

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 [16] is based on: polychrony. In the timing model of polychrony [9], 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* pairs a tag $t \in \mathbb{T}$ and a value $v \in \mathbb{V}$,
- a *signal* $s \in \mathcal{S}$ maps *chain* of tags $\mathcal{C} \subset \mathbb{T}$ to values $v \in \mathbb{V}$,
- a *behavior* $b \in \mathcal{B}$ maps signal names $X \subset \mathcal{V}$ to signals,
- a *process* $p \in \mathcal{P}$ is a set of behaviors of same domain.

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 $\emptyset$. We say that a reaction r is concatenable to a behavior b, written $b \cdot^? r$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and $\max(\mathcal{T}(b)) < \min(\mathcal{T}(c))$. If $b \cdot^? c$ then the concatenation of c to b is defined by $(b \cdot c)(x) = b(x) \uplus r(x)$ for all $x \in \mathcal{V}(b)$.

Synchronous structure: A behavior c is a *stretching* of time in a behavior b, written $b \leq c$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and there exists a bijection f on tags s.t.
$\forall t, u, t \leq f(t) \wedge (t < u \Leftrightarrow f(t) < f(u))$
$\forall x \in \mathcal{V}(b), \mathcal{T}(c(x)) = f(\mathcal{T}(b(x)))$
$\forall t \in \mathcal{T}(b(x)), b(x)(t) = c(x)(f(t))$
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 $q : I = \mathcal{V}(p) \cap \mathcal{V}(q)$.

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

Blocks

In Synoptic, we said that a block x receives control from its trigger and reset signals $x.\mathsf{trigger}$ and $x.\mathsf{trigger}$. To its interpretation corresponds a denotation parameterized by the time tags at which the trigger and reset events occur. These time tags are the chains C^t (for the trigger) and C^r (for the reset) and constrain the actual meaning of the block, defined by A.

$$[\![\mathsf{block}\ x\, A]\!] = \left\{ b \,|\, c \;\middle|\; \begin{array}{l} (b, c) \in [\![A]\!]^C \times \mathcal{B}|_{\substack{x.\mathsf{trigger} \\ x.\mathsf{reset}}} \\ C^r = \mathcal{T}(c(x.\mathsf{reset})) \\ C^t = \mathcal{T}_1(c(x.\mathsf{trigger})) \supseteq C^r \end{array} \right\}$$

Dataflow

A data-flow block $\mathsf{dataflow}\ f\, A$ is defined by the meaning of A controlled by the parent timing C.

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

An event $\mathsf{event}\, x \rightarrow y$ triggers x every time y occurs. Composition $A \,|\, B$ merges all timely compatible traces of A and B under the same context: $[\![A \,|\, B]\!]^C = [\![A]\!]^C \,|\, [\![B]\!]^C$.

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

An operation $\mathsf{data}\ y\, f\, z \rightarrow x$ assigns the product of the values u and v of y and z by the operation f to the signal x.

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

A delayed flow $\mathsf{data}\ y\, f\, z \rightarrow x$ assigns v to the signal x upon reset $t \in C^r$ and the previous value of y otherwise, at time $\mathrm{pred}_{C^x}(t)$.

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

Actions

The execution $c_k = [\![A]\!]_b$ of an action A is given a state b to return 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.

$$[\![\,\mathsf{do}\,A]\!]_b = b \cdot c \, where \, c_k = [\![A]\!]_b$$

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 $t = \max(\mathcal{T}(b)) = \min(\mathcal{T}(c))$.

$$[\![A; B]\!]_b = \begin{cases} (c \cdot d)_l, & k = 1 \\ (c \bowtie d)_l, & k = 0 \end{cases} \quad \begin{pmatrix} c_k = [\![A]\!]_b \\ d_l = [\![B]\!]_{b \cdot c} \end{pmatrix}$$

A conditional evaluates either $\langle A \rangle_b$ or sqB_b depending on the current value of x denoted by $b_{\uparrow x}$

$$[\![\,\mathsf{if}\,x\,\mathsf{then}\,A\,\mathsf{else}\,B]\!]_b = \begin{cases} \langle A \rangle_b, b_{\uparrow x} \\ \langle B \rangle_b, \neg b_{\uparrow x} \end{cases}$$

A skip does nothing but to return status 1 while an event defines x at some time $t > \max \mathcal{T}(b)$, as required by latter concatenation.

$$[\![\,\mathsf{skip}\,]\!]_b = \emptyset_1 \quad [\![x!]\!]_b = \{(x, t, 1)\}_0$$

Similarly, an operation $x = y \, f \, z$ defines x at $t > \max \mathcal{T}(b)$ by taking the current values $b_{\uparrow y}$ and $b_{\uparrow z}$ of y and z

$$[\![x = y \, f \, z]\!]_b = \{((x, t, f(b_{\uparrow y}, b_{\uparrow z}))\}_0$$

Automata

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: $\forall t \in C^r$, $b_i(s)(t) = 0$.

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

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.

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