LTSs for Translation Validation of (multi-clocked) SIGNAL specifications

Julio C. Peralta, Thierry Gautier, Loïc Besnard, Paul Le Guernic
INRIA Centre Rennes-Bretagne Atlantique, IRISA/CNRS, Campus Universitaire de Beaulieu, 35042 Rennes cedex, France
{*Julio.Peralta, Thierry.Gautier, Loic.Besnard, Paul.LeGuernic*}*@irisa.fr*

***Abstract*—Design of critical embedded systems demands for guarantees on the reliability of the implementation/compilation of a specification. In general, this guarantee takes either the form of a certified compiler, or the validation of each translation. Here we adopt the translation validation approach. In particular, we translate both the SIGNAL specification and the associated C simulator into LTSs. Then, an appropriate (successful) preorder test between both LTSs can be interpreted as a *refinement* between the C implementation and its source SIGNAL specification, otherwise, counter-examples are generated automatically. The feasibility of our approach is shown through examples.**

***Keywords*-Labelled Transition Systems; Multi-clocked Synchronous Programs; Concurrent Programs; Refinement**

I. INTRODUCTION

Design of critical embedded systems demands for guarantees in the reliability of the implementation/compilation from a specification. In general, this guarantee takes either the form of a certified compiler, or the validation of each translation. On the one hand, compiler certification proposes to verify each phase of the compiler. This approach has the drawback of freezing the potential improvements and/or developments of the compiler, given that a change in the compiler incurs the risk of redoing (part of) the verification. Translation validation, on the other hand, takes a common representation for the source and the target and compares them appropriately to decide on the presumed relation of refinement between the source and the target models. As opposed to compiler certification, validation of the translation has to be done for each source and target whereas a certified compiler is guaranteed to produce (at least) a refinement for every translation. Here we adopt the translation validation approach for specifications given in SIGNAL and its generated C simulators. In particular, we translate both the SIGNAL (multi-clocked) specification and the associated C simulator into LTSs. Then, an appropriate (successful) preorder test on both LTSs can be interpreted as a *refinement* between a C implementation and its source SIGNAL specification, otherwise, counter-examples are generated automatically. The feasibility of our approach is shown through examples.

The paper is organized as follows. Related work is revised in Section II. Section III introduces SIGNAL and FIACRE which are the languages used in our validation proposal to describe the source specifications and common LTS representation, respectively. We assume basic knowledge of the C programming language (the language of the implementation), and thus we do not present this language here. In Section IV a translation from SIGNAL to LTSs is provided, and then we present our correct implementation relation in the context of LTSs. Next, in Section V examples of translation from SIGNAL to LTSs are provided together with the translation of their generated simulators, in order to test their potential refinement relation. We conclude in Section VI and give references to future work. Finally, the appendix provides our algorithm to translate C to FIACRE (LTS textual representation).

II. RELATED WORK

A. Pnueli et al. [1], [2] were the sole attempts to validate the SIGNAL to C translation. They used so-called *synchronous transition systems* (STS) as a common representation for the specification and the implementation. STS are later interpreted as Boolean functions manipulated through BDD representations; and their refinement test amounts to matching selected states, comprising the values of input-output-memory variables, for the source and the implementation.

By contrast, our proposal is finer (exposing concurrency within an STS "state") and equally permits test of refinement modulo input-output-memory values, or a restriction to input-output values, depending on the desired granularity of the model representation. The choice of granularity does not affect the soundness of the refinement relation established, only its scalability. The scope of their translation validation is limited to the class of endochronous SIGNAL programs, whereas we do not impose such a restriction. It is not clear whether they restrict their validation for reasons of tractability or by lack of applicability of their method(s). Apparently, the reason is that endochronous programs enjoy some determinism properties that disappear once this class is abandoned. In addition, we do not use a Boolean representation of the specification, but an LTS representation, which may represent some savings in (an explicit) state representation. Nonetheless, our work was most inspired by this research.

On a different approach to validate the translations from embedded system specifications, G. Singh [3] translates Bluespec System Verilog specifications into PROMELA (the language used by model checker SPIN) as well as its RTL

implementation. The problem of refinement checking is recast as that of having both PROMELA models (the one obtained from the specification and the one obtained from its implementation) hold for the same set of LTL specifications. It is clear that the precision of this refinement test relies on the set of LTL specifications which depend themselves on the user capturing enough requirements about the models in the set of LTL formulas provided. In our translation validation approach we can also adopt this approach of refinement with respect to a set of temporal formulas, with the advantage that we are not restricted to LTL expressivity, but allow the greater expressivity of alternation-free regular mu-calculus [4].

III. SIGNAL AND FIACRE

A. The SIGNAL language

SIGNAL is a data-flow relational language that relies on the polychronous model [5], [6].

1) Introduction to SIGNAL*:* In SIGNAL, a process P consists of the parallel composition of equations $x :=op(y,z)$ over signals x, y, z defined on discrete time, where *op* is one of four kernel operators as explained next. A delay equation $x := y\,\$1\ \mathsf{init}\, v$ defines x every time/instant y is present. Initially, x is defined by the value v, and then, it is defined by the previous value of y. The unary delay is naturally extended to an N-ary delay in the equation $x := y\,\$\mathsf{N}\ \mathsf{init}\, v_1, \ldots, v_N$. A sampling equation $x := y\,\mathsf{when}\, z$ defines x by y when z is true (the simplified equation $x := \mathsf{when}\, z$ is equivalent to $x := z\,\mathsf{when}\, z$). A merge equation $x := y\,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise. Finally, 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 $P\,\mathsf{where}\, x$ restricts the signal x to the lexical scope of P.

$$P, Q ::= x := op(y,z) \mid P\,\mathsf{where}\, x \mid P \,|\, Q \qquad \text{(process)}$$

In SIGNAL, the presence of a value along a signal x, which represents the clock of x, is an expression noted $\hat{}x$, or $\hat{x}$. It is true, at a given instant, when x is present. Otherwise, it is absent. Specific derived processes and operators are defined in SIGNAL to manipulate clocks explicitly. We only use the simplest one, $x\hat{} = y$, that synchronizes all occurrences of the signals x and y (x and y are said synchronous).

2) Static analysis of SIGNAL *specifications:* Table I shows the *clock relations* associated with each primitive construct of SIGNAL. For the undersampling construct, the clock of the Boolean signal `b` is partitioned into $[b]$ and $[\neg b]$. The sub-clock $[b]$ (resp. $[\neg b]$) denotes the set of instants where the Boolean expression `b` is present and *true* (resp. *false*). Clock relations are automatically collected and inferred by the compiler from any program to be analyzed. For a process `P = P1|...|Pn`, its associated clock relations are the result of applying the clock calculus on the conjunction of the clock relations associated with the sub-processes `Pk`, $k \in 1..n$.

Table I
CLOCK RELATIONS FOR PRIMITIVES

construct	clock relations
`y := f(x1,...,xn)`	$\hat{y} = \hat{x}_1 = ... = \hat{x}_n$
`y := x $1 init c`	$\hat{y} = \hat{x}$
`y := x when b`	$\hat{y} = \hat{x} \cap [b]$, $[b] \cup [\neg b] = \hat{b}$ and $[b] \cap [\neg b] = \emptyset$
`z := x default y`	$\hat{z} = \hat{x} \cup \hat{y}$

Table II
SCHEDULING RELATIONS FOR PRIMITIVES

construct	scheduling relations
`y := f(x1,...,xn)`	$\hat{y} : \mathtt{x}_1 \rightarrow \mathtt{y}, \ldots, \quad \hat{y} : \mathtt{x}_n \rightarrow \mathtt{y}$
`y := x $1 init c`	
`y := x when b`	$\hat{y} : \mathtt{x} \rightarrow \mathtt{y}, \quad \hat{y} : \mathtt{b} \rightarrow \hat{y}$
`z := x default y`	$\hat{x} : \mathtt{x} \rightarrow \mathtt{z}, \quad \hat{y} - \hat{x} : \mathtt{y} \rightarrow \mathtt{z}$

Table II shows the scheduling relations associated with each primitive construct of SIGNAL. A conditional scheduling relation: $c : x \rightarrow y$ means that for all instants in the clock c the computation of y cannot be performed before the value of x is known. That is, the precedence is effective at the intersection of the three clocks: $\hat{x}, \hat{y}, c$. For any signal x, we have that the value of x cannot be computed before $\hat{x}$, the clock of x, hence the implicit scheduling relation $\hat{x} : \hat{x} \rightarrow x$. Notice that the delay does not have scheduling relations between its input and output. The composition of SIGNAL equations induces the union of their associated scheduling relations that we call the conditional dependency graph. Reasoning about this graph in conjunction with the information of the clock relations helps the compiler in producing a normalised SIGNAL program where the presence/absence of each program signal may be (uniquely) determined. This rewriting is key to resolve some nondeterminism in our proposed translation from endochronous SIGNAL specifications to LTSs.

More precisely, in order to assess the consistency of the clock relations associated with a program, and to organize the control of such a program, the so-called *clock calculus* of the compiler synthesizes a *clock hierarchy* [7], [8]. A clock hierarchy is a *dominates* relation on the quotient set of signals, according to $\hat{}=$ relation (x and y are in the same class iff they are synchronous, i.e. $x\hat{} = y$). Briefly, a class C dominates a class D (C *is higher than* D) if the clock of D is computed as a function of Boolean signals belonging to C and/or to classes dominated by C. When the clock hierarchy has a unique highest class, the process has a fastest clock (relative to the other clocks), called *master clock*, and there is a unique (deterministic) way to compute the clocks of all signals, we say that the process is *endochronous*. Such

FcrPgrm	→	*Dcls Prcs* `root_prc`
Prcs	→	*Comp Prcs* | *BPrc Prcs* | *BPrc*
Comp	→	`id`[´ *PortList* `]´ is` *LclPortDclList ParallelCmps*
BPrc	→	`id`[´ *PortList* `]´ is` *StateList* [`var` *VarDclList*] *InitStmt TransList*
PortList	→	*IdList* `:` *Type* | *IdList* `:` *Type*, *PortList*
StateList	→	`states` *IdList*
VarDclList	→	ϵ | *IdList* `:` *TypeVar*, *VarDclList*
TransList	→	*FromState Stmt TransList* | *FromState Stmt*
Stmt	→	*SelStmt*`;` *Stmt* | *IfStmt*`;` *Stmt* | *ToState*`;` *Stmt* | *CommStmt*`;` *Stmt* | *AssignStmt*`;` *Stmt* | ϵ
CommStmt	→	`id` `!´` *ExpList* | `id` `?´` *ExpList Cond* | `id`
Cond	→	ϵ | `where` *bool_exp*
SelStmt	→	`select` *Stmt NDetChs* `end`
NDetChs	→	`[]´` *Stmt* | `[]´` *Stmt NDetChs*

Figure 1. Subset of FIACRE language

```
1)  process fifo1[ p_x,p_sx:bool, c:none ] is
2)  states s1, s2, s3
3)  var rx1, b, C_sx, x, sx: bool
4)  init  rx1 :=  false; b := false; to s1
5)  from s1
6)      select
7)              b := not b;
8)              C_sx := not b;
9)              if (b) then
10)                 p_x ?x
11)             end;
12)             to s2
13)     []
14)             c;
15)             to s1
16)     end
17) from s2
18)     if (b) then rx1 := x end;
19)     if (C_sx) then
20)             sx := x;
21)             p_sx !sx
22)     end;
23)     to s3
24) from s3
25)     c;
26)     to s1
```

Figure 2. `fifo1` implementation in FIACRE

a program can be run in an autonomous way (its master clock plays the role of an activation clock). Otherwise, the program needs extra information from its environment to be run in a deterministic way.

B. The FIACRE language

In the following we present the syntax of the FIACRE language (resembling CSP [9]) which is one of the input languages for the tool-set CADP [10]. We will use CADP functionalities of model checking and preorder testing on LTSs, in order to prove refinement between the LTSs generated from our FIACRE programs. In particular, we need a textual representation (as FIACRE programs) of LTSs to describe the translation from a generated C program to an LTS.

An extract of FIACRE syntax is shown in Fig. 1 for further reference. Also, the FIACRE program (fragment) shown in Fig. 2 will serve as a running example in the explanation below, and for the first translation validation case.

Roughly speaking, a FIACRE program (*FcrPgrm*) consists of a set of global declarations (*Dcls*) for constants, types and channels, a set of basic processes (*BPrc*) out of which components (*Comp*, which are processes themselves) are built through parallel composition (*ParallelCmps*) with possible synchronization on their ports (*PortList*, *LclPortDclList*). (Fig. 2 shows one basic process named `fifo1` whose declaration spans lines `1-26`; its ports are `p_x`, `p_sx` and `c`, that communicate two Booleans and no data, respectively.) Clearly components serve to describe/construct models hierarchically without specifying any real computation, whereas basic processes (*BPrc*) do contain the actual computation (*TransList*) specified as a set of transitions in a textual description (from which an LTS is produced). This way we can associate a (computation) state with a basic process, by naming the state (label) it is in (from its *StateList*, line `2` in the example) and the value of its local variables (*VarDclList*, shown in line `3`). Initially, a basic process state is determined after executing its initialisation statement (*InitStmt*, and line `4` of our example program). Then, by (synchronous) composition of the (computation) state of its sub-components, we can inductively construct the (computation) state of a component and continue similarly until we reconstruct the (computation) state of the designated root process/component (`root_prc`).

A FIACRE transition begins by a designation of the state label from which (*FromState*, and lines `5`, `17` or `24`) its statements (*Stmt*, and line ranges `6-16`, `18-23`, `25-26`) can be executed, typically leading to other states through an unconditional jump (*ToState*, and lines `12`, `15`, `23`, `26`). The statements executed upon entrance to a transition up to the jump to another state are considered as an atomic step of the process where they occur. As a result, a string in concatenation *FromState Stmt* may denote more than one (LTS) transition if *Stmt* contains branching constructs (*SelStmt* and/or *IfStmt*). A statement is either a selection statement (*SelStmt*, as in lines `6-16`), an if-statement (*IfStmt*), an unconditional state transition statement (*ToState*), a port communication statement (*CommStmt*, shown in lines `10`, `14`, `21`, `25`[1]), or an assignment statement (*AssignStmt*, exemplified by lines `7`, `8`, `20`). Statements may be composed sequentially using `;´`. Most constructs (*IfStmt*, *AssignStmt*, *ToState*) have the standard imperative semantics, except the selection and communication statements that we explain next. A selection statement offers a non-deterministic choice (*Stmt* separated by`[]´`) between its (top) statements. A port communication statement (*CommStmt*), in turn, may be of any one of three sorts: emission (line `21`), (conditional) reception (line `10`), or dataless synchronization (lines `14`, `25`). The value(s) communicated, in port communication, are that(those) described by the (list of) expression(s) (*ExpList*) that follow the designated port. If more that one value is communicated, then we use a comma-separated list (of

[1]Recall that dataless synchronization uses no emission/reception symbol.

expressions) to communicate each value. For an emission, the value communicated depends on the value of the expression to be emitted, whereas for a reception the value received is the one that matches the expression given. When a variable is used for reception then all values in its type are possible. The condition on a reception (*Cond*) may impose extra constraints (*bool_exp*) on the values that can be received. Such a constraint is restricted to the syntax of standard Boolean expressions. Last, but not least, it is worth mentioning a syntactic restriction on the allowed occurrences of communication statements. *At most one* communication statement (emission or reception) is allowed per FIACRE process transition (that is, in any one execution path from the given state to a target state). In our example of Fig. 2 we can identify compliance with this restriction by the occurrence of at most one communication statement per branch of a select statement (lines `10`, `14`), and only one communication statement for transitions from states `s2`, `s3`, notably those in lines `21`, `25`, respectively. This restriction will be important for the discussion presented in Sect. V while extracting FIACRE models from C programs.

We will not present the compilation of FIACRE into LTS but refer the interested reader elsewhere [11] for a full description of the language semantics.

C. Labelled Transition Systems (LTSs)

In the following we will introduce the definition of an LTS, hiding of labels in LTSs, and the relations of (strong) bisimulation/equivalence and simulation between LTSs that we use for testing refinement.

As a way of notational convention suppose that $\mathcal{A}$ is a set of symbols called *observable actions*, and $\tau \notin \mathcal{A}$ the *unobservable action*. Given $A \subseteq \mathcal{A}$, we write A_τ for the set $A \cup \tau$.

Definition 1 (Labelled Transition System, LTS*):* An LTS is a quadruple $S = \langle Q, A, T, q_0 \rangle$, where Q is the set of *states*, $A \subseteq \mathcal{A}$ is the set of *observable actions*, $T \subseteq Q \times A_\tau \times Q$ is the *transition relation*, and $q_0 \in Q$ is the *initial state*.

We will write $q_1 \xrightarrow{a} q_2 \in T$ to denote that $(q_1, a, q_2) \in T$. We will use (sparingly) the parallel composition operator of two LTSs [9], say S_1 and S_2, that we note as "$S_1 \; |[A]| \; S_2$" to model the concurrent execution of S_1 and S_2 with forced synchronization on (action list) A.

Next, we present the hiding operator, which renders unobservable some otherwise observable actions.

Definition 2 (Action set hiding in an LTS*):* Let LTS $S = \langle Q, A, T, q_0 \rangle$, and let $B \subseteq \mathcal{A}$. The expression "hide B in S" denotes the LTS $\langle Q, A \setminus B, T', q_0 \rangle$ where T' is defined by the rules:

$$\frac{q \xrightarrow{a} q' \in T \;\wedge\; a \in B}{q \xrightarrow{\tau} q' \in T'} \qquad \frac{q \xrightarrow{a} q' \in T \;\wedge\; a \notin B}{q \xrightarrow{a} q' \in T'}$$

Observe that action $a \in \mathcal{A}$ above is assumed a constant string, and that communicating LTSs passing data are commonly denoted using the port name, say `x`, through which data, `D` for instance, is communicated; also, the symbols `!` or `?` are used to denote emission or reception. In principle, reception may be made on any value of the type of data that is declared for the designated port, but a variable of the same type could be used. In the multi-way synchronization scheme, that we use, several emissions and/or receptions could be matched when their port is the same and the data communicated is equal too. However, in order to compose in parallel two LTSs we assume receptions and emissions match iff the port and the data match regardless the sign (`?`, or `!`) used for communication. Moreover, for conciseness we will use a list of ports without listing the actual data when referring to synchronization sets in hiding and parallel composition.

Finally, consider the (strong) equivalence between two states of two LTSs which will be used to define (strong) equivalence of their respective LTSs.

*Definition 3 (*LTS *state equivalence relation, $\mathcal{R}_{eq}$):* Let two LTSs $S_i = \langle Q_i, A, T_i, q_{0_i} \rangle$ with $i = 1, 2$. Given $p \in Q_1$ and $q \in Q_2$ then p is *equivalent* to q, noted $p \; \mathcal{R}_{eq} \; q$, iff the two conditions below hold

(I) $\forall p \xrightarrow{a} p' \in T_1 \; \exists q \xrightarrow{a} q' \in T_2 \;\wedge\; p' \; \mathcal{R}_{eq} \; q'$;
(II) $\forall q \xrightarrow{a} q' \in T_2 \; \exists p \xrightarrow{a} p' \in T_1 \;\wedge\; q' \; \mathcal{R}_{eq} \; p'$

Hence two LTSs are *equivalent* iff their initial states are state equivalent (i.e. $q_{0_1} \; \mathcal{R}_{eq} \; q_{0_2}$). Moreover, we say that S_1 *simulates* S_2 iff the initial states of each LTS are related through a state simulation relation, noted $q_{0_1} \; \mathcal{R}_{imp} \; q_{0_2}$, where $\mathcal{R}_{imp}$ is defined by condition (I) alone (replacing $\mathcal{R}_{eq}$ by $\mathcal{R}_{imp}$). Similarly we can say that S_2 simulates S_1 iff $q_{0_2} \; \mathcal{R}_{imp} \; q_{0_1}$ and this time we use only condition (II) of the $\mathcal{R}_{eq}$ relation (with the appropriate replacement). Note that $\mathcal{R}_{eq}$ is an equivalence relation whereas $\mathcal{R}_{imp}$ is a preorder.

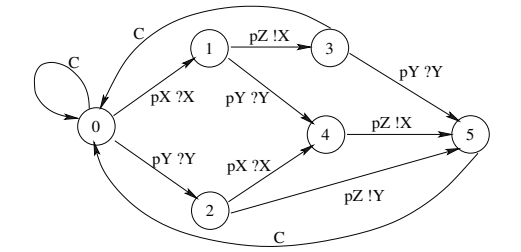
IV. FROM SIGNAL TO LTSS

Let us now describe our translation of kernel SIGNAL equations into LTSs [12]. For ease of readability and compactness LTSs are presented here graphically rather than textually (using FIACRE). However, a textual representation should be straightforward from the pictures provided.

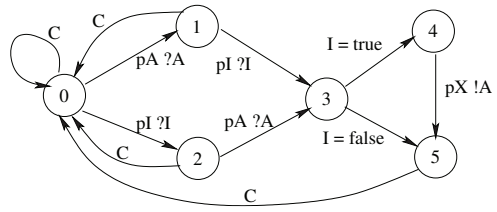
Briefly, for every source SIGNAL variable, say `X`, the translation uses a communication port named `pX` and the data communicated through such a port has the same name and type as the source SIGNAL variable, `X`, that it denotes. Additionally, the translation will assume a dataless port `C` whose occurrence, inside the generated LTS, denotes the end of an instant/reaction (according to SIGNAL semantics and the source program translated).

Observe that the actions of an LTS are (abstractly) represented as

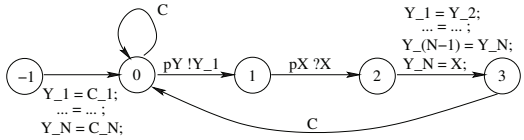
- `pX?X` for `pX` a port and `X` the expression or variable denoting the data received through that port; or



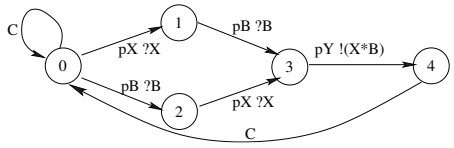
(a) `Z := X default Y`



(b) `X := A when I`



(c) `Y := X$N init C_1,...,C_N`



(d) `Y := X*B`

Figure 3. Abstract LTSs for SIGNAL kernel equations

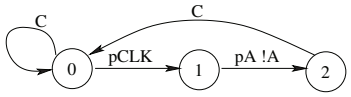


Figure 4. LTS for `CLK ^= A`

- `pX!X` for `pX` a port and `X` the expression or variable (whose current value is) emitted through that port; or
- assignments or Boolean expressions involving variables (and possibly constants).

Intuitively, a transition labelled with a Boolean expression will trigger (and thus occur) if and only if the action (denoting a Boolean expression) evaluates to true. The actual LTS actions are obtained when variables are replaced by their values, all communication offers are turned into emissions, and transitions labelled with true Boolean expressions or assignments merge their source state into their target state when executed/traversed. Transitions with Boolean expressions evaluating to false are removed.

An LTS for each of SIGNAL kernel equations is depicted in Fig. 3. All LTSs except the one of delay, Fig. 3(c), have label `0` in their initial state. The initial state of the LTS for a delay equation is labelled with `-1`. In practice, an instance of this LTS would have this state eliminated and its transition to `0` too, as explained above, given that the transition label is an expression that does not involve communication. As a result we can consider that the initial state of all possible instances of delay is labelled with `0`. For the LTSs of most operators we can appreciate that the possible ordering of receptions-emissions are dictated by considering all possible clocks of the SIGNAL operators. The delay operator is the exception, because one of the two (possible) orderings of the visible actions (emit-receive or receive-emit) induces deadlocks in composition. The implicit assumption for the whole translation is that the values are generated in the left-hand-side of a SIGNAL equation, hence the translation as emission, and the right-hand-side (of the same equation) receives the values generated elsewhere, thus the receptions, all within one instant/reaction. We use action `C` to mark the end of an LTS instant/reaction. Also, occurrences of `C` can be thought of as denoting a tick of a clock, an upper bound of all the clocks in any SIGNAL program.

Notice that all these transition systems (Fig. 3) are deterministic. Moreover, a trace of an elementary path/cycle in one of our (proposed or generated) LTSs from state with label `0` to state with label `0` represents a "synchronous" reaction. Two signals are "synchronous" iff either each (simple) cycle (from `0` to `0`) contains the label of both of them, or none of them.

In order to render the translation semantics preserving [12] we need the scheduling relations information from the source SIGNAL program, namely, the class of synchronous signals and also the reorganization of the source program such as that produced by SIGNAL clock calculus. Our LTS interpretation of a class of synchronous signals, say `CLK ^= A`, is given in Fig. 4. This is enough to express clock equivalences with more signals. That is, the clock equivalence expressed as `CLK^=A^=B^=C` would be translated as the parallel composition of three LTSs—one for each pair `CLK^=A`, `CLK^=B`, and `CLK^=C`—with synchronization on `CLK` and added `C`, i.e. LTS_1 `|[pCLK,C]|` LTS_2 `|[pCLK,C]|` LTS_3.

Parallel composition in SIGNAL is then replaced by parallel composition of the associated LTSs using the common variables as synchronization set. That is, given two LTSs, $S_i = \langle Q_i, A_i, T_i, q_{0_i} \rangle$ with $i \in \{1, 2\}$, generated by our translation, we compose them in parallel synchronizing on the common port names, $A = A_1 \cap A_2$: $S_1 | [A] | S_2$.

SIGNAL variable hiding is similarly translated as hiding in LTSs, following the mapping from SIGNAL variable names to LTS port names. For instance, SIGNAL process P `where` x will translate as `hide pX in` LTS_P. (Viewed as a FIACRE program, source signal `x` becomes a local port of some component.) The LTS first effect of hiding is that of replacing the set of actions listed as hidden by τ labels. We eliminate such transitions through $\tau^\star$`.a` bisimulation

```
process fifo1 = (? boolean x; ! boolean sx;)
  (| sx := current_1(x,^sx)
   | interleave(x,sx) |)
where
  process current_1 = (? boolean wx; event c;
                       ! boolean rx;)
    (| rx1 := wx default rx2
     | rx2 := rx1$1 init false
     | rx1 ^= wx ^+ when c       %clock addition%
     | rx := rx1 when c
     |) where boolean rx1, rx2; end;
  process interleave = (? boolean x, sx; !)
    (| x ^= when b
     | sx ^= when (not b)
     | b := not(b$1 init false)
   |) where boolean b; end;
end;
```

Figure 5. A one-place FIFO in SIGNAL

```
1)  logical x, sx;   /* input/output signals */
2)  logical rx1, b, C_sx; /* local signals */

3)  logical fifo1_initialize()    {
4)      rx1 = FALSE;
5)      b = FALSE;    }
6)  logical fifo1_iterate()    {
7)      b = !b;
8)      C_sx = !b;
9)      if (b)
10)         if (!r_fifo1_x(&x)) return FALSE;
11)     if (b) then rx1 := x end;
12)     if (C_sx) {
13)         sx = x;
14)         w_fifo1_sx(sx); }    }
```

Figure 6. C implementation of `fifo1`

reduction [13] since they do not change the input-output relation of the SIGNAL source, which is the focus of our interest for refinement checking. Also, refinement testing in the presence of τ transitions in the specification and the implementation LTSs is bound to fail given the optimized form of the implementation wrt the specification. Moreover, the presence of τ transitions in any one of the two LTSs composed in parallel may render our translation from SIGNAL to LTS an overapproximation for exochronous programs with clock constraints [12]. Endochronous programs, by contrast, do not rise this semantics preservation issue.

V. VALIDATION EXAMPLES

In the following we provide the source SIGNAL programs and assume the associated/corresponding LTS generated from the result of clock calculus on the given source program. Moreover, we present the reduced LTS where all τ transitions have been eliminated.

A. A one-place FIFO

Consider the SIGNAL program in Fig. 5 denoting a one-place FIFO. The input is a Boolean `x`, and the output is available, through signal `sx`, exactly one instant (in the clock of `b`) after the data has been input.

Now consider the generated C program (in Fig. 6) from the `fifo1` SIGNAL program. In general, the structure of the generated code executes in a (non-terminating) loop (not shown here). The first thing that it does is to initialize needed variables, i.e. the variables referring to a delay, in function `fifo1_initialize`, and then call the function `fifo1_iterate` to calculate a reaction of the program. Reading and/or writing of (input and/or output) signal values is preformed through functions `r_fifo1_signalname` and `w_fifo1_signalname`, respectively.

Now, the translation (Fig. 2) into FIACRE proceeds by creating one process with one port (and one local variable) for each input and output C-variable (so designated from the code generator of SIGNAL); the type of such ports are the type of the variable they denote. Designated local C variables become FIACRE local process variables. An extra dataless communication port `C` is added, in a similar way as was done for the translation from SIGNAL source, in order to mark the boundaries of a reaction. The initial action of such process corresponds to translating the body of C function `fifo1_initialize`, followed by a jump to the designated first state of such process (typically named `s1`). There is only one FIACRE statement that describes all possible transitions from the first state of the process: a `select` statement (*SelStmt*). The first choice offers the possible executions of the source C iterate function up to an i/o statement (typically a reading statement). The second, and last choice of the `select` statement is a transition that is supposed to let time pass without a reaction of the program (a stuttering step), which is not included in the C program for simplicity, determinism, and because C programs are not meant to be composed as it is done for SIGNAL programs. We need the stuttering transitions, however, for comparing specifications and implementations, and also to mark reactions boundaries of the implementation.

As regards the generation of the remaining states and transitions of the FIACRE `fifo1` process (Fig. 2), we follow a similar pattern as that briefly explained above: partition the control-flow of the C iterate function, creating blocks (of C statements) that end in an i/o C statement, and then translate each C block as a FIACRE transition. Because i/o in the C implementation is translated as port communication in the generated FIACRE process we had to split control-flow of the source C to produce valid FIACRE code compliant with the restriction of at most one communication statement per transition. FIACRE unconditional jump statements will serve to link FIACRE transitions following the control flow of the source C code. Assignments, conditionals, and expressions are quite similar in syntax and semantics between C and FIACRE, hence we will omit their discussion. The algorithm for this translation is given in the appendix where we use the well-known compilers concept of basic blocks as the partitioning criterion for simplicity of exposition.

The LTS associated with SIGNAL specification of Fig. 5 is given in Fig. 7. And the LTS for the implementation (rendered as a FIACRE program) in Fig. 2 is also in Fig. 7. Clearly, the relation between the implementation and the

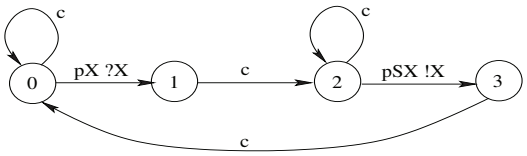


Figure 7. LTS for `fifo1` specification

```
process fifo2 = (? boolean x; ! boolean ssx;)
  (| ssx := fifo1(sx)
   | sx := fifo1(x) |)
where
  boolean sx;
  process fifo1 ... % Same code as in Fig. 5 %
```

Figure 8. Two-place FIFO in SIGNAL

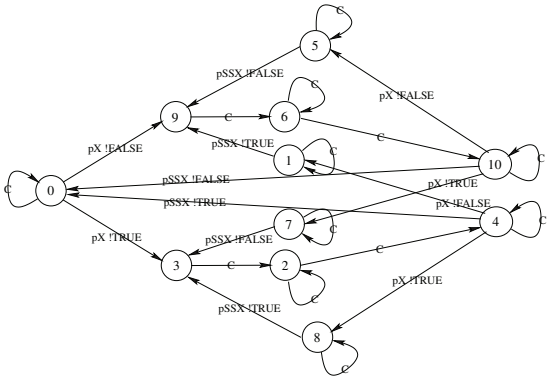


Figure 9. LTS for two-place FIFO

specification is indeed a refinement. It is important to notice here that our LTSs are minimal modulo $\tau^\star.\mathtt{a}$ reduction, otherwise the comparison may be too detailed to succeed. That is, τ transitions of the specification may not correspond to τ transitions of the implementation, and vice versa. Typically the implementation LTS contains less transitions/traces than the specification. However, this example does not exhibit this characteristic. Our next example will illustrate this difference between specification and implementation.

B. A two-place FIFO

Consider now a two-place FIFO specification in SIGNAL, as shown in Fig. 8. It re-uses the `fifo1` specification (Fig. 5), by composing two instances of the one-place FIFO to obtain a two-place FIFO. The novelty of this example is twofold. After clock calculus we learn that it has two master clocks and that there is a clock constraint unresolved by the compiler. Most notably, this clock constraint arises because there are multiple (two in this case) definitions for one event, and the compiler is unable to tell whether the definitions are equivalent. The event concerned here is the point of communication between the two one-place FIFOs. There are two definitions for this event because each FIFO reads and writes at the pace of its internal clock. Given that there are two instances of the one-place FIFO, there are two (seemingly) independent activation clocks that must meet in a specific event: the passing of data from the first to the second FIFO.

LTS *of specification:* The compositional LTS generation from this specification does reflect the cases where the clock constraint holds and where it does not; we have verified [12] using model-checking that this is indeed the case. However, if we compose the two LTSs denoting instances (as shown in Fig. 9) of a one-place FIFO, the resulting LTS only contains cases where the clock constraint is verified, hence our insistence on eliminating τ transitions.

Clock constraints in implementation: Generating a C program (Fig. 10) from this specification reflects the unresolved clock constraint (lines `31`, `32`) by *modifying the interface* of the implementation (signals `C`, `C2` in line `2`) with respect to that of the specification, thus exposing two (previously local) clock signals that (roughly) correspond to the clocks of the roots of the clock trees in the hierarchy produced by the clock calculus. The reason for the extended interface should be clear when one considers a C sequential implementation of a parallel source specification (in SIGNAL). Had the implementation language been parallel too, there would be no need for an interface extension; nonetheless such a parallel implementation would still contain some form of exception triggering in case of possible misuses. Returning now to the sequential implementation, there is the potential of an exception triggered by a violation of the clock constraint. Here the compiler (explicitly) distrusts the user by attaching the code to consider a possible misuse; alternatively, the compiler can obviate/skip this test if asked explicitly. In the end, either option would generate the same code differing only in the existence of the lines that trigger the exception. For the purposes of this exposition we will use the implementation that generates exceptions in case of a constraint violation.

Exochronous implementations: The reason that the implementation exposes two (previously internal) clock signals is that the specification is exochronous, as opposed to endochronous specifications where given the state of the program the clock of all its signals is uniquely determined (modulo stuttering steps). For exochronous programs, the compiler synthesizes a small controller on top of the original clock trees and asks the user to provide the clocks of the added root signals in each extended tree. This way the implementation is able to encode the reading of the data associated with the signals on the root of both trees.

Reasoning about clock constraints: Our translation into FIACRE will change slightly, in this example, for two reasons. We do not translate exceptions, and reading of some inputs is now done in one FIACRE communication event, as opposed to reading in separate events. These two changes are related in the sense that we search to produce an LTS where no exception situation can possibly arise. This amounts to

```
1)  /* ==> input/output signals */
2)  logical x, ssx, C, C2;
3)  /* ==> local signals */
4)  logical b, X1, b2, Csx, Csx2, b3, Cssx;

5)  logical fifo2_initialize()  {
6)     x = FALSE;
7)     b = FALSE;
8)     X1 = FALSE;
9)     b2 = FALSE;
10)    Csx = FALSE;  }
11) logical fifo2_iterate()  {
12)    if (!r_fifo2_C(&C)) return FALSE;
13)    if (!r_fifo2_C2(&C2)) return FALSE;
14)    if (C) {
15)      b = !b;
16)      Csx = !b;
17)      if (b)
18)        { if (!r_fifo2_x(&x)) return FALSE; }
19)    }
20)    Csx2 = (C ? Csx : FALSE);
21)    if (C2) {
22)      if (Csx2)
23)        X1 = x;
24)      b2 = !b2;
25)      Cssx = !b2;
26)      if (Cssx) {
27)        ssx = X1;
28)        w_fifo2_ssx(ssx); }
29)    }
30)    b3 = (C2 ? b2 : FALSE);
31)    if ((Csx2) != (b3))
32)       polychrony_exception("Csx2 != b3" );
33)    Csx = FALSE;
34) }
```

Figure 10. C implementation of `fifo2`

```
12)    if (!r_fifo2_C(&C0)) return FALSE;
13)    if (!r_fifo2_C2(&C20)) return FALSE;
14)    if (C0) {
15)      b1 = !b0;
16)      Csx0 = !b1;
19)    }
20)    Csx20 = (C0 ? Csx0 : FALSE);
21)    if (C20) {
24)      b21 = !b20;
29)    }
30)    b30 = (C20 ? b21 : FALSE);
31)    if ((Csx20) != (b30))
32)       polychrony_exception("Csx20 != b30" );
```

Figure 11. Slice of `fifo2` implementation

back-propagating the condition of exception to the reading of the clocks added to the interface. We will proceed by computing a backward slice [14] of the `fifo2_iterate` function, from the site where the exception is potentially triggered. This produces a program (Fig. 11) consisting of lines 12, 13, 14, 15, 16, 19, 20, 21, 24, 29, 30, 31 and 32, where the last two lines concern the reading of the clocks that interest us. Next, we label the occurrences of all variable references (in the slice) with an index giving the C code a static single-assignment form [15], as shown in Fig. 11. The interest of having this form is to make explicit the (variable) definition that a (variable) occurrence is referencing. It should not be surprising that the pass to SSA form is so neat (without introducing the ϕ-functions typical of SSA form), since SIGNAL is declarative, and the `iterate` function of the C code does not contain cycles, except when arrays are used in the SIGNAL specification. We can now reason about the assignments and conditionals as Boolean expressions. In particular, we are interested in propagating backwards (from line `31` to lines `12`, `13`) the expression denoting the *compliance* with the clock constraint, i.e. the negation of the expression `((Csx2`$_0$`) != (b3`$_0$`))`. This goal may be achieved by successive (forward) substitutions of the Boolean equations associated with each statement until we have expressed the variables referenced in the clock constraint in terms of variables defined at the moment of reading the clocks in lines `12`, `13`. Consequently, replacing `b`$_1$'s definition by its sole reference in line `16` yields equation `Csx`$_0$ `= !(!b`$_0$`)`; then replacing this definition in line `20` results in equation `Csx2`$_0$ `= C`$_0$ `&& b`$_0$ which is ready for its use in the reading statement of `C`$_0$, given that it references variables in the same scope. Proceeding in a similar way for `b3`$_0$ we obtain the equation `b3`$_0$ `= C2`$_0$ `&& (!b2`$_0$`)`. Finally, the result of complying with the clock constraint `(Csx2`$_0$ `== b3`$_0$`)` can be recast as the equivalent constraint `(C`$_0$ `&& b`$_0$`) == (C2`$_0$ `&& (!b2`$_0$`))`. It is worth noticing that this new constraint can be enforced if we know the values of clocks `C`$_0$ and `C2`$_0$ *at the same time*, provided that the values of the other variables are normally accessible. Considering the new form of the relational constraint above we will change our translation for reading the (extra) clock variables of the FIACRE interface. Briefly, we need to declare a FIACRE channel to communicate a pair and then declare the appropriate port (in the port list of a process declaration) as communicating using the declared channel. The FIACRE program in Fig. 12 shows the translation of the C implementation in Fig. 10, where we have enforced the compliance with the clock constraint, and hence we do not have to worry about capturing in the translation the part dealing with possible violation of the clock constraint (though we include it for completeness of the translation). Only two small modifications of our algorithm (shown in the appendix) are needed to produce FIACRE code after the processing explained above: (i) specialise the basic block decomposition to avoid splitting the reading of new interface clocks into different blocks; and (ii) translation of the sequential composition for reading of such clock signals, since they will always occur together.

Testing refinement: We now seem to be in a position to test refinement between the SIGNAL specification and its C implementation. However, the LTS for the implementation (not shown) has an extra port, and associated events, that are not visible (neither exist) in the specification: the port named `p_cc` of Fig. 12. It suffices to hide such port (and associated events) and to minimize the resulting LTS modulo $\tau^\star$`.a` reduction, so as to eliminate the τ transitions generated by hiding. Here, unlike our one-place FIFO refinement test (Sect. V-A), the LTS of the implementation and that of the specification (Fig. 9) are different. The actual test determines

```
channel TC is bool#bool
process fifo2[p_cc:TC, p_x,p_ssx:bool, c:none] is
states s1, s2, s3, s4
var C, C2, b, b2, b3, Csx, Csx2, X1, Cssx, ssx,
    x: bool
init x := false; b := false; X1 := false;
     b2 := false; Csx := false; to s1
from s1
  select
    p_cc ?C,C2 where (C and b) ≡ (C2 and (not b2));
    to s2
  []
    c; to s1
  end
from s2
  if (C) then
        b := not b;
        Csx := not b;
        if (b) then p_x ?x end
  end;
  to s3
from s3
  if (C) then Csx2 := Csx
         else Csx2 := false end;
  if (C2) then
      if (Csx2) then X1 := x end;
      b2 := not b2;
      Cssx := not b2;
      if (Cssx) then ssx := X1; p_ssx !ssx end
  end;
  to s4
from s4
  if (C2) then
      b3 := b2
  else
      b3 := false
  end;
  if (Csx2 <> b3) then ... end;
  Csx := false;
  c;
  to s1
```

Figure 12. Implementation of `fifo2` in FIACRE

whether a (strong) simulation relation between the implementation and the specification exists: the result is positive. The implementation indeed simulates the specification and hence it is a refinement. It is worth noting here that the difference between the two LTSs is that the implementation imposes an input (communication through port `p_X`) before any output (communication through port `p_SSX`) is available, even if this is possible in the same instant. The LTS for the specification, by contrast, exhibits the two possible orders of (instantaneous) input and output whenever they are possible. This difference is automatically generated by the CADP functionality for testing equivalence between two LTSs modulo (strong) equivalence/bisimulation.

C. Validation using input-output-memory values

Here we only mention, for lack of space, that our validation approach using LTSs also allows the possibility to test a finer correspondence between the C implementation and the SIGNAL specification. Notably, we will now test the refinement relation using input-output-memory values [2] as visible/observable actions in our generated LTSs, as opposed to the restricted visibility of input-output actions we considered in our previous examples.

As a sort of comparison between the LTSs exposing input-output observations and those for input-output-memory observations, for the one-place FIFO example (Sect. V-A) the state space of the specification LTS goes from 4 states and 6 transitions to 22 states and 35 transitions in the detailed specification, whereas for the implementation we had 4 states and 6 labels that raised to 15 states and 20 transitions. The refinement test is still valid, as expected. However, in this case the LTS for the specification is not equivalent to the LTS of the implementation, as was the case for a restricted observation. The reason for this difference being that the specification allows several interleavings (within one reaction) of the events associated with memory variables and those associated with input and output, while the implementation LTS considers only one (possible) ordering. This result is most natural, since our translation of SIGNAL into LTSs exposes the concurrency of events arising within reactions. Had we used a state-based representation, as Pnueli [2], this parallelism would be hidden, while our action-based representation does the opposite.

The same experiment, of allowing observations of memory values in addition to input-output values, for the two-place FIFO was also successful. The sources and results of all our experiments can be found at ftp://ftp.irisa.fr/local/signal/publis/SIG2C_TV/.

VI. Concluding Remarks

We have shown a translation of C programs generated from (multi-clocked) SIGNAL specifications into LTSs (via a FIACRE textual representation). Given that SIGNAL specifications can be translated into LTSs too, we are able to test the possibility of a (strong) simulation relation between the LTS of the C implementation and the LTS of the SIGNAL specification. Given the detail of the comparison and that of the representation we argue that the success of this test can be interpreted as a refinement relation between the implementation and the specification, otherwise a counter-example is generated automatically. Other preorder relations on LTSs exist (e.g. those implemented in CADP toolset). In particular, testing for the weak trace simulation relation on our LTSs (for specification and implementation) amounts to establishing the "trace refinement" relation used in CSP [9]. We show through examples the feasibility of our translation validation approach, provided models are finite state.

We have also shown some rudiments of reasoning about unresolved clock constraints (or assertions) in (multi-clocked) SIGNAL implementations, in order to extend the reasoning capabilities of the SIGNAL compiler, and as a compile time code manipulation/transformation. That this technique is extensible to all C programs (generated by the SIGNAL compiler) is the subject of our current research. Nonetheless, we anticipate that this (however schematic) technique can be extended to (SIGNAL and the generated C) programs with scalar domains.

Last, but not least, the translation from C programs to LTSs, in the context of the SIGNAL code generator, can be regarded as a model extraction technique that is exploitable through model checking in CADP [10].

Future work: We envisage to construct an automatic model extractor (from C to FIACRE) for the C programs generated by the SIGNAL compiler, in order to test the scalability issues of our proposed technique. Moreover, this model extractor should be easily extensible and applicable to the SIGNAL generated C code that uses threads, which is another application avenue for our translation validation proposal. As regards extraction of finite state models from SIGNAL and/or C programs, we do not discard the adoption of existing techniques in this respect, most notably those based on counter-example guided abstraction refinement [16], and/or those [17] based on bounding the reasoning about the values of program variables, for deciding equivalence.

REFERENCES

[1] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Proceedings of TACAS'98*. LNCS 1384, 1998, pp. 151–166.

[2] A. Pnueli, O. Shtrichman, and M. Siegel, "Translation validation: From SIGNAL to C," in *Correct System Design Recent Insights and Advances*. LNCS 1710, March 2000, pp. 231–255.

[3] G. Singh and S. K. Shukla, "Verifying compiler based refinement of Bluespec^{TM} specifications using SPIN model checker," in *15th International SPIN Workshop on Model Checking Software*. LNCS 5156, 2008, pp. 250–269.

[4] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free mu-calculus," *Sci. Comp. Program.*, no. 46, pp. 255–281, 2003.

[5] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *Journal of Circuits, Systems, and Computers*, vol. 12, no. 3, pp. 261–304, 2003.

[6] L. Besnard, T. Gautier, and P. Le Guernic. (2010, March) SIGNAL V4-INRIA version: Reference Manual. [Online]. Available: http://www.irisa.fr/espresso/Polychrony/

[7] T. P. Amagbegnon, L. Besnard, and P. Le Guernic, "Arborescent canonical form of Boolean expressions," INRIA/IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France, Tech. Rep. 2290, 1994.

[8] T. P. Amagbegnon, L. Besnard, and P. Le Guernic, "Implementation of the dataflow synchronous language SIGNAL," in *Proceedings of PLDI'95*. ACM Press, 1995.

[9] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[10] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2006: A toolbox for the construction and analysis of distributed processes," in *CAV'07*. Springer LNCS 4590, 2007, pp. 158–163.

[11] B. Berthomieu, J.-P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stocker, and F. Vernadat, "The syntax and semantics of FIACRE," March 2009, deliverable No. 4.2.4 of project ANR05RNTL03101 OpenEmbeDD.

[12] J. C. Peralta, T. Gautier, P. Le Guernic, and L. Besnard, "Labelled transition systems for compositional description of multi-clocked SIGNAL specifications," submitted for publication.

[13] R. Mateescu, "On-the-fly state space reductions for weak equivalences," in *FMICS'05*. ACM, 2005, pp. 80–89.

[14] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel, "A formalisation of the relationship between forms of program slicing," *Sci. Comput. Program.*, vol. 62, no. 3, pp. 228–252, 2006.

[15] M. M. Brandis and H. Mössenböck, "Single-pass generation of static single-assignment form for structured languages," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1684–1698, 1994.

[16] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav, "Efficient verification of sequential and concurrent C programs," *Form. Methods Syst. Des.*, vol. 25, no. 2-3, pp. 129–166, 2004.

[17] Y. Rodeh and O. Strichman, "Building small equality graphs for deciding equality logic with uninterpreted functions," *Inf. Comput.*, vol. 204, no. 1, 2006.

APPENDIX

Here we briefly outline the (unoptimized) algorithm to translate the C programs generated from SIGNAL (multi-clocked) specifications, into a FIACRE process. Translation of assignments and branching statements is straightforward because the names of the SIGNAL source and C variables remain the same. The only C source statements that will be dealt with differently are the input, output, and exception invocations. Input and output will be translated as reception and emission, respectively, using ports having the name of the signal used for reading/writing. Exception invocations may be translated as jumps to a special blocking state, for thoroughness of the translation, even though we proposed one way to translate such C programs so that this transition never arises.

As a preprocessing of the input C program we propose a traversal of the `iterate` function to construct a suitable control-flow graph, CFG, from which FIACRE code generation is simple. Such a traversal constructs a standard CFG for the given function, where vertices are labelled by basic blocks of the source. Clearly statements labelled by input, output and/or exception invocation will occur at most once in a basic block. As a side processing of this CFG construction we add a special label to each vertex to denote a different state of the FIACRE process that will be constructed. As a result, we have a labelled directed graph $G = \langle V, E \rangle$ with set of vertices V and edges E. Each vertex of V has two labels: (a) one denoting a FIACRE state label, noted `state`(v); and (b) one denoting the C statements of a basic block, noted `statements`(v), for $v \in V$. Also, we will refer to the sole entry vertex of G as v_i, and an extra vertex $v_o \notin V$ will be needed to "close" the transitions of the generated FIACRE process using a synchronization on port `C`.

Input: Initialization function and $G = \langle V, E \rangle$ of `iterate` function
Output: A FIACRE process declaration, P_F

1: *Translate contents of initialization function into the* `init` *section of* P_F, *and translate the* `return` *statement as a jump statement to* `state`(v_i)
2: *Declare one port and local variable for each input/output source variable, and add port* `C`
3: *Declare a local variable for each source local variable*
4: $\forall v \in V \cup \{v_o\}$ *declare state* `state`(v)
5: **for all** $v \in V \wedge v \neq v_i$ *create transition text* **do**
6: *Declare start of transition:* `from state`(v)
7: **for all** $s \in$`statements`(v) *translate as* **do**
8: **if** s *an assignment* **then**
9: *translate as assignment*
10: **else if** s *a conditional* **then**
11: *translate as conditional and fill-in true and false branches with a jump statement to successor states* `state`(v_t) *and* `state`(v_f), *respectively (i.e.* $v \xrightarrow{\texttt{true}} v_t, v \xrightarrow{\texttt{false}} v_f \in E$)
12: **else if** s *an input or output statement* **then**
13: *translate as port communication followed by jump to successor state* `state`(v')
14: **else if** s *a* `return` **then**
15: *translate as jump to state* `state`(v_o)
16: **else if** s *an* `exception` *statement* **then**
17: *translate as jump to artificial deadlock state* `state`(v_x)
18: **end if**
19: **end for**
20: **end for**
21: *Add a transition text:* `from state`(v_o) `C; to state`(v_i)
22: *Add select statement:* `from state`(v_i) `select` *with translation of* `statements`(v_i) *as in lines 6-16 above, and with alternative branch containing* `C; to state`(v_i) `end`.