A Refinement-based compiler development for synchronous languages

Jean-Paul Bodeveix, M Filali, Shuanglong Kan

A Refinement-based compiler development for synchronous languages

Jean-Paul Bodeveix
IRIT Université Paul Sabatier
Toulouse, France
bodeveix@irit.fr

Mamoun Filali-Amine
IRIT CNRS
Toulouse, France
filali@irit.fr

Shuanglong Kan
Nanjing University of Aeronautics and Astronautics
Nanjing, China
kanshuanglong@outlook.com

ABSTRACT

In this paper, we are concerned by the elaboration of generic development steps for the code generation for synchronous languages. Our aim is to provide a correct by construction solution. For that purpose, we adopt a refinement-based approach where proof obligations for each step guarantee properties preservation. We use the Event-B formal method. We start with a big step semantics specified by an Event-B machine. Through a sequence of refinements, expressed as Event-B refinement machines, we end up with a code generation step which implements a small step semantics preserving the properties of the big step semantics.

KEYWORDS

synchronous languages, refinement, code generation, semantics, verification

1 INTRODUCTION

Our study concerns system development. We are interested by the development of reactive systems. For such systems, for instance, avionic systems and interactive systems, synchronous languages have revealed well suited. From the beginning, synchronous languages have put forward a strong mathematical framework underlying their basic principles: synchrony and deterministic concurrency [5]. Thanks to that, synchronous languages have received a significant audience, especially within the critical embedded systems community where high levels of certification are required. However, one must acknowledge that although significant efforts have been devoted to explain the software architecture for implementing frameworks for the development of such languages and even to mechanize them [6, 10, 19], the production of a certified compiler remains a challenge.

https://doi.org/10.1145/3127041.3127056

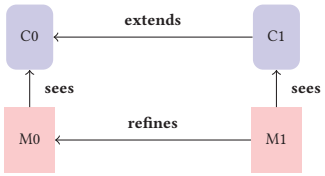


Figure 1: Event-B development step

In this paper, we are concerned by the elaboration of generic development steps for the code generation for synchronous languages. Our aim is to provide a correct by construction solution. For that purpose, we adopt a refinement-based approach where proof obligations for each step guarantee the preservation of some classes of properties [7]. We use the Event-B formal method. We start with a big step semantics specified by an Event-B machine. Through a sequence of refinements, expressed as Event-B refinement machines, we end up with a code generator and an interpreter which implements a small step semantics preserving the properties of the initial big step semantics, i.e. the sequences of reactions observed when executing the concrete machine were allowed by the big step semantics.

The rest of the paper is organized as follows: in the next section, we give an overview of the Event-B language. In Section 3, we present the development of a generic compiler for synchronous languages. Finally, in Sections 4 and 5, we discuss related and future work.

2 EVENT-B

The Event-B method allows the development of correct by construction systems and software [1]. To achieve this, it supports natively a formal development process based on a refinement mechanism with mathematical proofs. Figure 1 illustrates a refinement step where a machine M0 using a context C0 (the **sees** edge) is refined (the **refines** edge) by a machine M1 using an extension C1 of C0 (the **extends** edge). Contexts define abstract data types through sets, constants and axioms while machines define symbolic labelled transition systems. The state of a transition system is defined as the value of machine variables. Labelled transitions are defined by events specifying the new value of variables while preserving invariant properties. Proof obligations for wellformedness and invariant preservation are automatically generated by the Rodin tool [18].

They can be discharged thanks to the available automatic proof engines (CVC4, Z3, …) or through human-assisted proofs.

2.1 Notations

For the most part, Event B uses standard set theory and its usual set notation. As a matter a fact, in Event-B, an array and a function are both considered as set of couples. Some notations are specific to Event B :

- **pair construction:** pairs are constructed using the maplet operator $\mapsto$. A pair is thus denoted $a \mapsto b$ instead of (a, b). The set of pairs $a \mapsto b$ where $a \in A$ and $b \in B$ is denoted $A \times B$.
- A subset of $A \times B$ is a *relation*. A relation r has a domain : $\mathbf{dom}(r)$ and a codomain : $\mathbf{ran}(r)$. When a relation r relates an element of $\mathbf{dom}(r)$ with at most one element, it is called a function. The set of partial functions from A to B is denoted $A \nrightarrow B$, the set of total functions is denoted $A \rightarrow B$. The image of a set A by a relation r is denoted $r[A]$. Last, the relational composition is denoted by the ; infix operator. Suppose $f \in A \rightarrow B$ and $g \in B \nrightarrow C$, then $f;g \in A \nrightarrow C$ is the following set of pairs:$\{x \mapsto z \mid \exists y.\ y \in B \wedge x \mapsto y \in f \wedge y \mapsto z \in g\}$.
- **domain restriction:** $D \lhd r = \{x \mapsto y \mid (x \mapsto y) \in r \wedge x \in D\}$
- **overwrite:** $f \lhd\!\!\lhd g = ((\mathbf{dom}(f) \backslash \mathbf{dom}(g)) \lhd f) \cup g$.
  For instance, such a notation is used to denote a new array obtained by changing the element of an array A at index i: $A \lhd\!\!\lhd \{i \mapsto e'\}$.
- **lambda expressions:** the usual expression for a lambda expression is: $\lambda x.\ x \in T \mid bdy$, where T is a set denoting the "type" of x: here the definition domain and bdy is the body of the lambda expression.

As already said, Event-B machines specify symbolic transitions through events. An event has three optional parts: parameters (**any** p1 …pn), guards (**where** …) specifying constraints to be satisfied by parameters and state variables, and actions (**then** …) specifying state variables updates. Guards are defined in set-based predicate logic. Concurrent updates of distinct variables may be deterministic ($x := e$), non deterministic ($x :\in E$ or $x :|\ P(x, x')$). In $x :\in E$, x takes any value belonging to the set E. In $x :|\ P(x, x')$, the new value x' of x is specified by the predicate P.

2.2 Preliminary example

As a preliminary example for section 3.1, Figure 2 illustrates basic labelled transitions systems [4]. The static description of transition systems is given through the context `cLTS`. `cLTS` introduces `State` and `Label` as abstract **sets**. Then, the set of initial states is given through the **constant** $\mathcal{I}$, and the transition relation through the constant `Trans`. The description of the dynamics of a transition system is given through the machine `mLTS`. First, we represent the state space by the variable `st` $\in$ `State`. The dynamics is specified through two events: the `INITIALISATION` event which chooses non-deterministically a state among elements of the set $\mathcal{I}$ and the `next` event which also chooses non-deterministically a label and a transition having such a label.

Figure 3 illustrates an Event-B refinement of the preceding machine `mLTS`. Intuitively, we model the introduction of an internal event that will take control for some time after which the `next` event

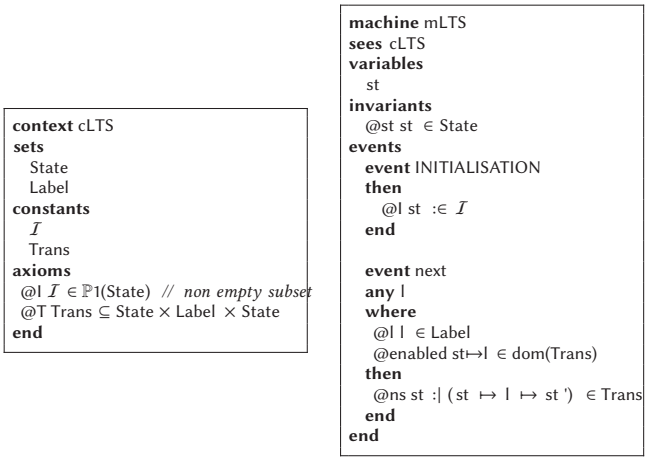


Figure 2: Event-B LTS semantics

becomes enabled again. For such a purpose, the machine `mLTS_r` **refines** `mLTS` and introduces a new variable `cpt` : a counter that will be initialized non deterministically. When the counter reaches the zero value, `next` is enabled. The refined machine refines the events `INITIALIZATION` and `next` and introduces the new event `silent` which is similar to τ events of CCS [12]. We notice that in the new (refining) `next` event, the `enabled` guard is strengthened by the conjunct $cpt = 0$. Moreover, this new `next` event provides a *witness* of the parameter of the old (refined) `next` event: the parameter `lr` is said to be the witness of the parameter `l`. When the new `next` event has been executed, `cpt` is again assigned a positive value. In Event-B, the proof obligations enforce a weak refinement semantics [12]. The purpose of the **variant** clause is to enforce that introduced events will not take control forever. Usually, the variant belongs to a well ordered domain and is decreased by newly introduced (internal) events declared to be **convergent**.

3 A GENERIC COMPILER DEVELOPMENT FOR SYNCHRONOUS LANGUAGES

In this section, we elaborate a generic refinement-based approach for the compilation of synchronous languages. Starting from the so-called labelled synchronous transition systems, we propose a sequence of refinement steps, illustrated by Figure 4, leading to code generation. Thanks to the refinement-based approach, the generated code is known to preserve by construction the initial semantics provided all the proof obligations have been discharged.

3.1 Labelled synchronous transition systems

In the synchronous context, executions are seen as sequences of reactions [17]. A reaction r assigns values to the subset of variables said to be present: undefinedness of r on a variable v means absence of v in r. We suppose a fixed domain $\mathcal{D}$ for the values and define a reaction over the set of variables V as a partial function from V to the domain $\mathcal{D}$, i.e., $\text{Reaction}(V) = V \nrightarrow \mathcal{D}$. We remark that the usual definition of a reaction is a total function to $\mathcal{D} \cup \{\bot\}$ where $\bot$

```
machine mLTS_r refines mLTS sees cLTS
variables st cpt
invariants
  @cpt_ty cpt ∈ ℕ
variant cpt
events
  event INITIALISATION
    then
      @l st :∈ ℐ
      @cpt_i cpt := 0
  end
  event next refines next
    any lr // parameter is renamed
    where
      @lr lr ∈ Label
      @enabled cpt = 0 ∧ st ↦ lr ∈ dom(Trans)
    with
      @l l = lr // link with abstract paramater; l witness relation ·
    then
      @ns st :| (st ↦ lr ↦ st') ∈ Trans
      @restart cpt :∈ ℕ
  end
  convergent event silent
    where
      @g cpt > 0
    then
      @cpt_d cpt := cpt − 1
  end
end
```

Figure 3: Event-B refinement

is a special symbol denoting absence. The use of a partial function is better suited to Event-B.

We introduce now labelled synchronous transition systems which generate synchronous executions.

Definition 3.1 (LSTS). A Labelled Synchronous Transition System is a quadruple $\langle \mathcal{V}, \mathcal{S}, \mathcal{I}, \rightarrow \rangle$ where $\mathcal{V}$ is a set of variables, $\mathcal{S}$ a set of states, $\mathcal{I} \in S$ is an initial state and $\rightarrow \subseteq \mathcal{S} \times \text{Reaction}(\mathcal{V}) \times \mathcal{S}$ is the set of transitions labelled by reactions.

3.1.1 Signal and its LSTS semantics. In order to illustrate the definition of an LSTS, we present the syntax of a kernel of the Signal language [8] and its semantics defined as an LSTS [17]. A Signal program over a set of variables V is a set of equations of the following form which define each variable at most once:

- $x_1 = x_2$ **when** x_3 *undersampling*
- $x_1 = x_2$ **default** x_3 *deterministic merge*
- $x_1 = f(x_2, x_3)$ *instantaneous function*
- $x_1 = x_2\$$ **init** v *delay*

With respect to the Signal language, we essentially do not consider local variables. This aspect is not relevant for our presentation. The semantics of Signal can be defined as an LSTS where a state is a mapping from memorized (or persistent) variables to values. A variable is memorized if it is introduced by a delay equation. In order to define the denotational semantics of Signal, we introduce the function R_m where m is a memory state. It takes as parameter a Signal equation and returns the set of allowed reactions. We also suppose the existence of a constant $\top \in \mathcal{D}$ representing true.

- $R_m[x_1 = x_2 \textbf{ when } x_3] =$
  $\{r \in V \nrightarrow \mathcal{D} \mid \forall v \cdot x_1 \mapsto v \in r \Leftrightarrow (x_2 \mapsto v \in r \wedge x_3 \mapsto \top \in r)\}$
- $R_m[x_1 = x_2 \textbf{ default } x_3] =$
  $\{r \in V \nrightarrow \mathcal{D} \mid \forall v \cdot x_1 \mapsto v \in r \Leftrightarrow (x_2 \mapsto v \in r \vee x_2 \notin \textbf{dom}(r) \wedge x_3 \mapsto v \in r)\}$
- $R_m[x_1 = f(x_2, x_3)] =$
  $\{r \in V \nrightarrow \mathcal{D} \mid \forall v_1, v_2, v_3 \cdot x_1 \in \textbf{dom}(r) \Leftrightarrow x_2 \in \textbf{dom}(r) \Leftrightarrow x_3 \in \textbf{dom}(r) \wedge (x_1 \mapsto v_1 \in r \wedge x_2 \mapsto v_2 \in r \wedge x_3 \mapsto v_3 \in r \Rightarrow v_1 = f(v_2, v_3))\}$
- $R_m[x_1 = x_2\$ \textbf{ init } v] =$
  $\{r \in V \nrightarrow \mathcal{D} \mid x_1 \in \textbf{dom}(r) \Rightarrow r(x_1) = m(x_1) \wedge x_1 \in \textbf{dom}(r) \Leftrightarrow x_2 \in \textbf{dom}(r)\}$

We are now ready to introduce the semantics of a Signal program P over variables of $\mathcal{V}$ as an LSTS. We define the set $\mathcal{M} \subseteq \mathcal{V}$ of memory variables as those introduced by $\$$ **init** equations. Then, the LSTS associated to P is the quadruple $\langle \mathcal{V},\ \mathcal{M} \rightarrow \mathcal{D},\ \{v \mapsto c \mid v = w\$ \textbf{ init } c \in P\}, \{m \xrightarrow{r} m' \mid r \in \bigcap_{p \in P} R_m[p] \wedge m' = m \lhd \{v \mapsto r(w) \mid w \in \textbf{dom}(r) \wedge v = w\$ \textbf{ init } c \in P\}\rangle$.

In the following, we will use an abstraction of the set of reactions $\bigcap_{p \in P} R_m[p]$ as a *parameterized* constraint problem. The parameters are here the variables which are not defined through a Signal program equation.

3.1.2 The refined LSTS. We intend to produce LSTS from synchronous languages as Lustre or Signal. The corresponding LSTS are specialized as follows:

- The reaction is computed from the synchronous program and the current state. For instance, in the Signal setting, the program can be read as a set of constraints of which solutions are the set of allowed reactions.
- The state contains the value of variables defined by a *delay* construct. The state space is thus a mapping from a (memory) subset $\mathcal{M}$ of the set of variables $\mathcal{V}$ to the domain $\mathcal{D}$.
- The state is updated by storing for each memory variable x its next value defined to be the value of some variable $\mathcal{N}(x)$ when x is present. Otherwise, the memorized value remains unchanged, thus allowing persistence of information over cycles.

Thus, we introduce what we call a refined LSTS (rLSTS) by the following definition:

Definition 3.2 (rLSTS). A rLSTS is a quintuple $\langle \mathcal{V}, \mathcal{M} \subseteq \mathcal{V}, \mathcal{I} \in \mathcal{M} \rightarrow \mathcal{D}, \mathcal{N} \in \mathcal{M} \rightarrow \mathcal{V}, \mathcal{C} \in (\mathcal{M} \rightarrow \mathcal{D}) \rightarrow \mathbb{P}(\mathcal{V} \nrightarrow D)\rangle$ where:

- $\mathcal{V}$ is a set of variables,
- $\mathcal{M}$ is the subset of memory variables,
- $\mathcal{I}$ is the initialization of these variables,
- $\mathcal{N}$ associates to each memory variable the variable, the value of which is stored,
- $\mathcal{C}$ returns the set of allowed reactions from a given memory state.

The LSTS corresponding to the rLSTS is defined as follows by the quadruple $\langle \mathcal{V} = \mathcal{V},\ \mathcal{S} = \mathcal{M} \rightarrow \mathcal{D},\ \mathcal{I} = \mathcal{I},\ m \xrightarrow{r} m' \equiv r \in \mathcal{C}(m) \wedge m' = m \lhd \mathcal{N}; r\rangle$ where $\mathcal{C}$ associates to a memory state a set of reactions.

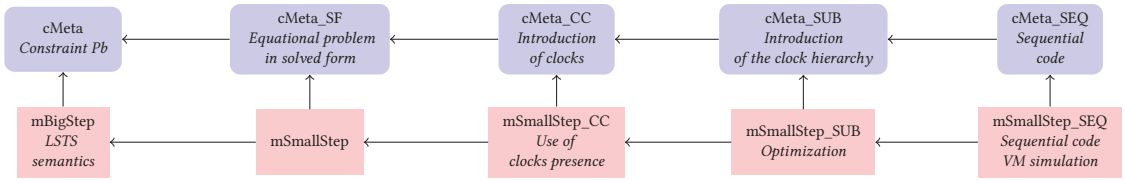


Figure 4: Refinement steps

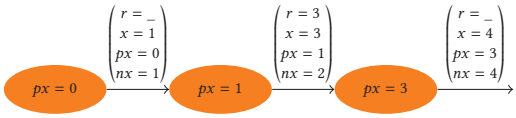


Figure 5: Signal LSTS partial run

3.1.3 Example. Consider the Signal program which increments `x` and restarts it from the signal `r` each time it occurs:

```
process INCR = (?integer r; !integer x;)
(|
  (|
   | px := x$ init 0
   | nx := px + 1
   | x := r default nx
   |)/px,nx
|)
```

If we keep local variables visible, the associated rLSTS is defined as follows:

$$\begin{aligned} \mathcal{V} &= \{r, x, px, nx\}, \\ \mathcal{M} &= \{px\}, \\ \mathcal{I} &= \{px \mapsto 0\}, \\ \mathcal{N} &= \{px \mapsto x\}, \\ C(m) &= R_m[px = x\$ \textbf{ init } 0] \\ &\cap\ R_m[nx = px + 1] \cap R_m[x = r \textbf{ default } nx] \end{aligned}$$

Figure 5 is a partial run of this example. A run is an alternating sequence of states (ovals) and transitions labelled by reactions (vectors). In the first reaction, `r` is absent and `x` is one more than the memorized value. The second transition is labelled by a reaction where `r` is present. Thus `x` takes the value of `r`. In the third transition, `r` is chosen to be absent and `x` is incremented. The reaction could also be empty (all variables being absent). However, this will be forbidden in the remaining of the paper to avoid Zeno behaviors.

3.1.4 Event-B representation. First of all, we define the context `cMeta` (Fig. 6) containing the various fields of the rLSTS we will consider. With respect to definition (3.2), we have added one parameter of type $\mathbb{P}(\mathcal{V})$ to the function C denoting the constraint problem of which reactions should be solutions. It contains the set of variables that are constrained and is now the full set $\mathcal{V}$. This parameter will be modified to specify the incremental computation of the reaction in the next refinement.

Second, we define the semantics of the given rLSTS through an Event-B machine which specifies the Labelled transition system associated to the rLSTS through the state variable `m` initialized using $\mathcal{I}$. The state evolves through the `bigStep` event parameterized by a reaction `r`. The reaction is accepted if it is a solution of the constraint system.

This `bigStep` event is intended to be split into small steps that compute reactions incrementally.

context cMeta
sets $\mathcal{V}$ $\mathcal{D}$
constants $\mathcal{M}$ $\mathcal{I}$ $\mathcal{N}$ C
axioms
  @fVat finite ($\mathcal{V}$)
  @$\mathcal{I}$_ty $\mathcal{I} \in \mathcal{M} \to \mathcal{D}$
  @$\mathcal{N}$_ty $\mathcal{N} \in \mathcal{M} \to \mathcal{V}$
  @C_ty $C \in (\mathcal{M} \to \mathcal{D}) \to (\mathbb{P}(\mathcal{V}) \to \mathbb{P}(\mathcal{V} \nrightarrow \mathcal{D}))$
end

Figure 6: The Event-B context cMeta

machine mBigStep **sees** cMeta
variables m // *memory*
invariants
  @m_ty m $\in \mathcal{M} \to \mathcal{D}$
events
  event INITIALISATION
  then
    @m_init m $:= \mathcal{I}$
  end
  event bigStep
  any r **where**
    @r_ty r $\in \mathcal{V} \nrightarrow \mathcal{D}$ // *r is a reaction*
    @g r $\in C$(m)($\mathcal{V}$) // *r is a solution of the constraint system*
    @ne r $\neq \emptyset$ // *to avoid Zeno behaviors*
  then
    @m m := m $\Leftarrow$ ($\mathcal{N}$;r) // *memory update*
  end
end

3.2 Solved form (context `cMeta_SF`) and small step semantics (machine `mSmallStep`)

This refinement aims at splitting the big step which specifies what are valid reactions. The small steps should compute one variable at a time. To make it possible, the constraint system should be put in solved form.

3.2.1 Solved form. We define the constraint problem as a set of equations supposed to be in triangular form: a left hand side variable is said to depend on right hand side variables and this dependency relation should be acyclic. In fact, two informations must be computed: the presence of a variable, and if present, its value. The value of a present variable is either defined by a formula, or read from input (indeterminate), or read from memory. Thus, we first introduce the constant `isPresent` together with the dependency relation `pdep`. The function `isPresent` takes as parameters a variable and a partially computed reaction and returns a set of

booleans which indicate if the variable is allowed to be present or absent, or both. `isPresent` returns either {TRUE}, either {FALSE}, or {TRUE,FALSE} to encode indeterminacy, as it is the case for master clocks of Signal programs [8]. Concerning pdep, it maps each variable to the set of variables on which its presence depends. Moreover, a variable which does not depend on other variables can be chosen to be present or absent.

constants isPresent pdep
axioms
@isPresent_ty isPresent $\in \mathcal{V} \times (\mathcal{V} \nrightarrow \mathcal{D}) \longrightarrow \mathbb{P}1(\mathbb{B})$
@pdep_ty pdep $\in \mathcal{V} \longrightarrow \mathbb{P}(\mathcal{V})$
@check_empty $\forall$ v · pdep(v) = $\emptyset \Rightarrow$ isPresent (v$\mapsto\emptyset$) = $\mathbb{B}$

Figure 7: cMeta_SF (part I)

Now, we introduce the eqns function for value computation, together with the dependency relation dep. Contrary to presence, values are specified in a functional way. Indeterminacy of values is inferred from the absence of equations for some variables. The function eqns takes as parameters the considered variable and returns its value for the currently computed reaction. This function is partial at several orders: it is defined for a subset of non memory variables. Accepted reactions (which are partial functions) should at most give a value to variables on which the first argument depends. These reactions are those that allow the presence of the variable.

constants eqns dep
axioms
@eqns_ty eqns $\in \mathcal{V} \nrightarrow ((\mathcal{V} \nrightarrow \mathcal{D}) \nrightarrow \mathcal{D})$
@mem_eqns $\mathcal{M} \cap$ dom(eqns) = $\emptyset$
@dep_ty dep $\in \mathcal{V} \longrightarrow \mathbb{P}(\mathcal{V})$
@dep_def $\forall$v· v $\in$ dom(eqns) $\Rightarrow$ dom(eqns(v)) $\subseteq$ (dep(v) $\nrightarrow \mathcal{D}$)

Figure 8: cMeta_SF (part II)

Figure 9 introduces the link between Presence and the existence of a value for a variable: a partial solution ps allows the presence of a variable v if and only if the partial solution ps is in the domain of eqns(v).

axioms
@isp_dom $\forall$v,ps· v$\in$dom(eqns) $\wedge$ ps $\in$ dep(v) $\nrightarrow \mathcal{D}$
$\Rightarrow$ (TRUE $\in$ isPresent(v$\mapsto$pdep(v) $\lhd$ ps) $\Leftrightarrow$ ps $\in$ dom(eqns(v)))

Figure 9: cMeta_SF (part III)

We suppose that for any variable v, pdep(v) is a subset of dep(v): the value of v (depending on variables of dep(v) can be computed once we know its presence (depending on variables of pdep(v). Then, we add properties stating that dep is transitively closed and irreflexive, thus a strict partial order.

Now, we define the constraint problem $C_{m,S}$ for a memory state m and a subset $S \subseteq \mathcal{V}$ of seen variables. A solution r is a partial function on S, undefinedness meaning absence. It is constrained by three properties (Fig. 11):

@pdep_dep $\forall$v· pdep(v) $\subseteq$ dep(v)
@pdep_trans $\forall$v1, v2· v1 $\in$ pdep(v2) $\Rightarrow$ pdep(v1) $\subseteq$ pdep(v2)
@dep_trans $\forall$v1, v2· v1 $\in$ dep(v2) $\Rightarrow$ dep(v1) $\subseteq$ dep(v2)
@dep_irrefl $\forall$v· v $\notin$ dep(v)

Figure 10: cMeta_SF (part IV)

- The presence or the absence of a variable v within a reaction r should be allowed by the set returned by the function isPresent where r is restricted to variables on which v depends.
- The restriction of r to memory variables is given by the memory state.
- The value of a present variable v associated to an equation is given by this equation taking as parameter v and a restriction of r to variables it depends on.

axioms
@C_def C =
(λm· m $\in \mathcal{M} \longrightarrow \mathcal{D}$ | λS· S $\subseteq \mathcal{V}$ |
{r | r $\in$ S $\nrightarrow \mathcal{D}$
$\wedge$ ($\forall$v· v$\in$S $\Rightarrow$ bool(v $\in$ dom(r)) $\in$ isPresent (v $\mapsto$ pdep(v) $\lhd$ r))
$\wedge$ (($\mathcal{M} \cap$ S) $\lhd$ r $\subseteq$ m)
$\wedge$ $\forall$v· v $\in$ dom(r) $\cap$ dom(eqns) $\Rightarrow$ r(v) = eqns(v)(dep(v) $\lhd$ r)
})

Figure 11: The constraint problem

3.2.2 Memory access. We introduce dedicated variables, from the set NextMem, used as markers for memory updates. The injective function write associates to each memory variable v, its memory write event. This event should be fired after all reads of v and after the computation of the next value of v given by the variable $\mathcal{N}(v)$. The presence of the write event depends on the presence of the next value. It is actually present if the next value is present.

constants NextMem write
axioms
@write write $\in \mathcal{M} \rightarrowtail$ NextMem
@wdep $\forall$v·v$\in\mathcal{M} \Rightarrow$ {w | v $\in$ dep(w)} $\cup$ {v, $\mathcal{N}$(v)} $\subseteq$ dep(write(v))
@wpdep $\forall$v·v$\in\mathcal{M} \Rightarrow$ pdep(write(v)) = {$\mathcal{N}$(v)} $\cup$ pdep($\mathcal{N}$(v))
@wpres $\forall$ps,v· ps$\in\mathcal{V}\nrightarrow\mathcal{D} \wedge$ v$\in\mathcal{M} \Rightarrow$
isPresent (write(v)$\mapsto$pdep(write(v)) $\lhd$ps) = {bool($\mathcal{N}$(v)$\in$dom(ps))}

3.2.3 Small step semantics. Given a constraint system in triangular form, it is possible to compute reaction variables one by one through so-called small steps. In the same way, the memory can be updated incrementally once the current value of a variable becomes useless in the current cycle: it will not be read anymore. For this purpose, we introduce a refinement machine with three new variables: the partially updated memory pm, the partial solution ps and the set of processed variables done.

machine mSmallStep **refines** mBigStep **sees** cMeta_SF
variables pm // *partially updated memory*
ps // *partial solution*
done // *processed signals*
invariants
@pm pm $\in \mathcal{M} \longrightarrow \mathcal{D}$
@ps_ty ps $\in \mathcal{V} \nrightarrow \mathcal{D}$
@done_ty done $\subseteq \mathcal{V}$

These variables satisfy the following invariant properties:

- The restrictions of pm and m to unprocessed memorized variables are identical functions.
- ps is a partial solution restricted to variables of done.
- If a variable is present, variables of which depends its *value* (given by dep) have been processed.
- If a variable has been processed, variables on which depends its *presence* (given by pdep) have also been processed.
- The partial solution is not empty as soon as one variable has been processed, i.e. the first processed variable should be marked as present.
- The memory is partially updated by the next value of memory variables of which the write event has been processed.

```
invariants
  @pmu ∀v·v∈𝓜\done ⇒ pm(v) = m(v) // pm equals m on unprocessed variables
  @ps_ok ps ∈ 𝐶(m)(done) // ps is the restriction to done of a reaction
  @seen_val ∀v·v ∈ dom(ps) ⇒ dep(v) ⊆ done
  @seen_pres ∀v·v∈done ⇒ pdep(v) ⊆ done
  @done_ne done ≠ ∅ ⇒ ps ≠ ∅ // forces non empty reactions
  @pm_m pm = m ⩤ (write⁻¹[done]◁𝒩; ps) // pm is a partial update of m
```

Initially, the partial solution and the set of processed variables are empty. Memory gets the rLSTS initial value.

```
  event INITIALISATION
    then
      @init_ps ps := ∅
      @init_done done := ∅
      @init_pm pm := ℐ
  end
```

Small steps determine the presence and compute the value of individual variables. These steps must be transparent from the point of view of the abstract machine: they do not modify previously introduced variables and should apply a finite number of times. This is why they are declared *convergent*, which means that they must decrease the *variant*. Several convergent events are defined. They respectively process absent variables, present variables defined by an equation, memory accesses or undetermined variables (considered as inputs or parameters). Due to space restrictions, two cases are presented: equational variables and absent variables. Other events are related to memory read and write variables, parameters (non memory and non equational variables).

The following listing manages variables defined by an equation. The event selects such a variable v which has not been processed before. Variables on which it depends should have been processed and v is allowed to be present. Then, its binding to the value given by the equation is inserted into the partial solution and v is added to done. It follows that the variant decreases.

```
variant 𝒱 \ done
events
  convergent event smallStep_P // processing of a present variable
    any v where
      @v v ∈ dom(eqns) \ done
      @d dep(v) ⊆ done // thus pdep(v) ⊆ done
      @p TRUE ∈ isPresent(v↦pdep(v) ◁ ps) // v is allowed to be present
    then
      @ps ps(v) := eqns(v)( dep(v) ◁ ps)
      @done done := done ∪ {v}
  end
end
```

The following listing manages absent variables. Contrary to present ones, they are processed by blocks. This will allow optimisations in the generated code. So, we consider a non empty set of variables V such as variables on which they depend are either already processed or in V. Absence should be allowed for all these variables. Note that this permission is checked by supposing variables of V are absent as unprocessed variables are seen as absent. Lastly, the partial solution should not be empty to ensure that the computed reaction will not be the empty one.

```
  convergent event smallStep_mA // processing of absent variables
    any V where
      @v V ⊆ 𝒱 \ done
      @vnv V ≠ ∅ // ensures that the variant decreases
      @d ∀v· v ∈ V ⇒ pdep(v) ⊆ done ∪ V
      // variables of V are allowed to be absent
      @abs ∀v· v ∈ V ⇒ FALSE ∈ isPresent (v↦pdep(v) ◁ ps)
      @ps ps ≠ ∅
    then
      @done done := done ∪ V
  end
```

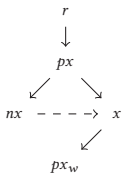
When all the variables have been processed by the small steps, we can refine the big step of the abstract machine and prepare the next round by resetting the variable done.

```
  event bigStep refines bigStep
    where
      @g done = 𝒱
    with
      @r r = ps // r witness relation
    then
      @ps ps := ∅
      @d done := ∅
  end
```

3.2.4 Example. Given the Signal example previously introduced in Paragraph 3.1.3, we add the write event marker px_w for the memory variable px. Figure 12 shows how the declared constants introduced in the solved form can be instantiated. The graph represents the functions pdep and dep. All the antecedents with plain arrows of a node x form its dependency for pdep. dep is obtained by adding the dashed arrows.

Concerning the `isPresent` function, written `isP` in the figure, it allows presence and absence for r. If r is actually present, px must be present; otherwise px presence does not care. Then the authorization of presence of variables nx, x and px_w is determined by the actual presence of px and x.

Concerning eqns, it associates a value to nx and x by accessing to the value of variables they depend on.



$$
\begin{aligned}
isP(r, ps) &= \{T, F\} \\
isP(nx, ps) &= \{px \in \mathbf{dom}(ps)\} \\
isP(x, ps) &= \{px \in \mathbf{dom}(ps)\} \\
isP(px_w, ps) &= \{x \in \mathbf{dom}(ps)\} \\
isP(px, ps) &= \{T\} \textbf{ if } r \in \mathbf{dom}(ps) \\
&\quad \textbf{else } \{T, F\}
\end{aligned}
$$

$$
\begin{aligned}
eqns(nx)(ps) &= ps(px) + 1 \\
eqns(x)(ps) &= ps(r) \textbf{ if } r \in \mathbf{dom}(ps) \\
&\quad \textbf{else } ps(nx)
\end{aligned}
$$

Figure 12: Solved Form level (example)

3.3 Introduction of clocks (machine mSmallStep_CC)

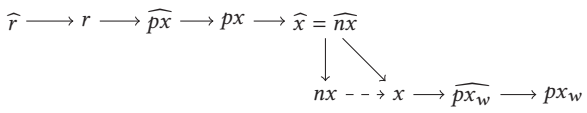
We now associate clocks to variables. The role of a clock is to manage presence. Thus, a variable will be present if and only if its clock is present. For this purpose, the set Clock is declared as a subset of variables supposed to be neither memory variables nor defined by an equation: they are not valued. Each variable has a clock (defined by the function cP). The clock of a clock is itself.

```
context cMeta_CC extends cMeta_SF
constants Clock cP
axioms
  @Clock_ty Clock ⊆ 𝒱 \ (ℳ ∪ NextMem ∪ dom(eqns))
  @isP_ty cP ∈ 𝒱 → Clock
  @isP_clk ∀s· s ∈ Clock ⇒ cP(s) = s
  @Clk_def ∀s·cP(s)=s ⇒ s∈Clock
  @isP_dep ∀s· s∈𝒱\Clock ⇒ cP(s) ∈ pdep(s)
  @clk_dep ∀v· v∈Clock ∧ dep(v) = ∅ ⇒ v ∈ IO
  @chk_clk ∀v,ps· ps∈𝒱↛𝒟
     ⇒ isPresent (v↦pdep(v)◁ps) = isPresent (cP(v)↦pdep(cP(v))◁ps)
  @pdep_clk ∀v· v∈Clock ⇒ pdep(v) = dep(v)
end
```

A new refinement can now be defined. It differs from the previous machine by the fact that presence of clocks are tested through the isPresent function while presence of valued variables comes to the presence of their clocks. One invariant is added. It states that for a processed variable, its presence and the presence of its clock are equivalent. The following listing illustrates the transformation of the two events. For equational variables: cP(v) presence is now checked. For absence, constraints on V are strengthen to ensure clocks are also selected and that they are absent. The big step is unchanged.

```
machine mSmallStep_CC refines mSmallStep sees cMeta_CC
variables pm ps done
invariants
  @pres ∀v· v ∈ done ⇒ (cP(v) ∈ dom(ps) ⇔ v ∈ dom(ps))
events
  event smallStep_P refines smallStep_P
    any v where
      @v v ∈ dom(eqns) \ done
      @d dep(v) ⊆ done
      @p cP(v) ∈ dom(ps)
    then
      @ps ps(v) := eqns(v)( dep(v) ◁ ps)
      @done done := done ∪ {v}
  end
  event smallStep_mA extends smallStep_mA
  where
      @clk_done cP[V] ⊆ done ∪ V
      @clk_abs cP[V] ∩ dom(ps) = ∅
  end
  event bigStep extends bigStep end // unchanged
end
```

3.3.1 Example. Given our working example, we introduce the clocks $\widehat{r}, \widehat{px}, \widehat{x}, \widehat{nx}, \widehat{px_w}$, thus defining the cP function by $cP(v) = \widehat{v}$. We have chosen to take the same clock for x and nx. This is allowed by the axioms since their respective presence are defined in the same way. The dependencies between the variables are now given by the following graph:



3.4 Clock hierarchy (machine mSmallStep_SUB)

The goal of this refinement is to introduce the clock inclusion relation: if a clock is absent, sub-clocks are also absent and it is not useful to check them at runtime. For this purpose, we introduce the relation sub represented by a function associating to a clock the set of clocks it subsumes (or contains). This relation should be compatible with the pdep relation: a clock depends on clocks containing it. Subsumption declarations should be consistent with presence semantics as given by the isPresent function.

```
context cMeta_SUB extends cMeta_CC
constants sub
axioms
  @sub_ty sub ∈ Clock → ℙ(Clock) // gives the set of subsumed clocks
  @sub_dep ∀v,w· w ∈ Clock ∧ v ∈ sub(w) ⇒ w ∈ pdep(v)
  @sub_present ∀v1,v2· v1∈Clock ∧ v2 ∈ sub(v1) ⇒
    (∀m,D,ps· m∈ℳ→𝒟 ∧ pdep(v1) ⊆ D ∧
     pdep(v2) ⊆ D ∪ {v1} ∪ sub(v1) ∪ cP⁻¹[{v1} ∪ sub(v1)] ∧
     v1 ∉ D ∧ sub(v1) ∩ D = ∅ ∧ ps ∈ 𝒞(m)(D) ∧
       FALSE ∈ isPresent(v1 ↦ pdep(v1) ◁ ps) ⇒
         FALSE ∈ isPresent(v2 ↦ pdep(v2) ◁ ps))
end
```

The refined machine declares one new invariant stating that if a clock has not been processed, the clocks it subsumes have not been dealt with either. The only modified event is the one managing absence: clocks declared to be absent are clocks subsumed by a clock v or variables of which clock is subsumed by v. Thus only one absence test is performed to discard V.

```
machine mSmallStep_SUB refines mSmallStep_CC sees cMeta_SUB
variables pm ps done
invariants
  @sub_done ∀c· c∈Clock\done ⇒ sub(c) ∩ done = ∅
events
  event smallStep_mA refines smallStep_mA
    any v V where
      @v v ∈ Clock \ done
      @d pdep(v) ⊆ done
      @abs FALSE ∈ isPresent(v↦pdep(v) ◁ ps)
      @ps ps ≠ ∅
      @V V ⊆ {v} ∪ sub(v) ∪ cP⁻¹[{v} ∪ sub(v)]
      @Vv v ∈ V
      @Vd V ∩ done = ∅
      @Vp ∀w· w ∈ V ⇒ pdep(w) ⊆ done ∪ V
    then
      @done done := done ∪ V
  end
end
```

3.4.1 Example. Considering our example, we add the subsumtion relation as bended arrows to the dependency graph restricted to clocks.

These new arrows define a subgraph of the dependency graph and should satisfy the axiom sub_present. For instance, let us consider the bended arrow between $\widehat{px}$ and $\widehat{x}$. Given a partial solution ps defined at least on $pdep(px)$ and not on px we must check the implication between each presence property. This property is satisfied since the right side of the implication is True because $px \notin \mathbf{dom}(ps)$.

3.5 Sequential code generation (machine mSmallStep_Seq)

This refinement step is the key point of the verified compiler generation. It consists in separating actions that can be performed at compile time (statically) from actions performed at run time, the link between them being the generated code. The principle is the following:

- At the abstract level, an event is fired if its parameters satisfy static properties (linked to the dependency relations) and dynamic properties (linked to presence / value) of variables in the partial reaction already built.
- To each such event is associated a convergent event that will select its parameter depending only on static properties and store it in the sequence of instructions built by the compilation phase. These events constitute the code of the compiler.
- The abstract event itself is refined by an event which simulates the created instruction by testing and updating the dynamic state.
- The correction of this separation relies on the fact that static properties should be sufficient to select event parameters. It is ensured if we suppose that, given parameters satisfying the static properties, there exists an event of which dynamic properties are also satisfied.

These principles are applied to small step events. Such an event may be fired on a variable if previous variables (for dep or pdep partial orders) have been processed and if some conditions depending on the partial reaction (presence of the clock for example) are satisfied. Building the reaction is left for the dynamic phase. However, variable selection, i.e., topological sort, can be performed at compile time. As a result, we get a sequence of statements specifying the event to fire and the variable to be selected. Statements are declared in the cMeta_Seq context:

context cMeta_Seq **extends** cMeta_SUB
sets Stmt
constants
GetPresence // *check clock presence*
Get$\mathcal{P}$ // *read parameter statement*
Read$\mathcal{M}$ // *read memory statement*
Write$\mathcal{M}$ // *write memory statement*
Assign // *assignment statement*
axioms
@gst **partition**(Stmt,{GetPresence},{Get$\mathcal{P}$},{Read$\mathcal{M}$},{Write$\mathcal{M}$},{Assign})
end

3.5.1 The compilation events. The compiler encodes machine instructions numbered from 0 to $N-1$ through two variables: stmt for the name of the statement and arg for the concerned variable. In order to take into account the clock hierarchy, we add a *jump* pointer (iff read if false) to each presence test. It indicates the continuation in case the test fails. So, we introduce the following variables:

machine mSmallStep_Seq **refines** mSmallStep_SUB **sees** cMeta_Seq
variables pm // *memory (dynamic, inherited)*
ps // *partial reaction (dynamic, inherited)*
N // *size of code (static)*
pc // *program counter (dynamic)*
stmt // *statements (static)*
arg // *argument of statement (static)*
iff // *forward pointer to follow in case of absence (static)*
invariants
@sta_N N ∈ ℕ
@dyn_pc pc ∈ 0. .N
@stmt_ty stmt ∈ dom(arg) → Stmt
@sta_arg arg ∈ 0. .(N−1) ⤖ $\mathcal{V}$
@iff iff ∈ stmt^{-1}[{GetPresence}] → 0. .N

The invariant specifies the link between old and new dynamic variables (done and pc) and between generated instructions and expected properties of the variable selected by the old small step events:

- processed variables (the set done) are arguments of previous statements in the code sequence,
- dependencies are arguments of previous statements
- arguments of GetPresence are clocks
- arguments of GetParam are parameters
- arguments of ReadMem are memorized variables
- arguments of WriteMem are memory write events
- arguments of Assign are equational variables
- iff pointers are forward pointers
- iff pointers skip subsumed clocks

invariants
@dyn_done done = arg[0. .pc−1]
@sta_dep ∀i· i ∈ dom(arg) ⇒ dep(arg(i)) ⊆ arg[0. .(i−1)]
@GetPresence_cstr arg[stmt^{-1}[{GetPresence}]] ⊆ $\mathcal{P}$ ∩ Clock
@Get$\mathcal{P}$_ctr arg[stmt^{-1}[{Get$\mathcal{P}$}]] ⊆ $\mathcal{P}$ \ Clock
@Read$\mathcal{M}$_ctr arg[stmt^{-1}[{Read$\mathcal{M}$}]] ⊆ $\mathcal{M}$
@Write$\mathcal{M}$_ctr arg[stmt^{-1}[{Write$\mathcal{M}$}]] ⊆ ran(write)
@Assign_ctr arg[stmt^{-1}[{Assign}]] ⊆ dom(eqns)
@iff_gt ∀i· i ∈ stmt^{-1}[{GetPresence}] ⇒ iff(i) > i
@iff_spec ∀i· i ∈ stmt^{-1}[{GetPresence}]
⇒ arg[i+1. .iff(i)−1] ⊆ sub(arg(i)) ∪ cP^{-1}[sub(arg(i))] ∪ cP^{-1}[{arg(i)}]

We now consider the event smallStep_P of paragraph 3.3. The management of other events is similar. According to the splitting method, we introduce a convergent event acting as the compilation step. It non-deterministically selects an equational variable provided that the variables it depends on have already been processed. Then it adds a new instruction, here a conditional assignment and updates the iff pointers to allow the new instruction to be skipped if the required conditions are satisfied.

convergent event sta_smallStep_P
any v **where**
@v v ∈ dom(eqns) \ ran(arg)
@d dep(v) ⊆ ran(arg)
then
@N N := N + 1
@arg arg(N) := v
@argss stmt(N) := Assign
@iff iff := iff ⩤
{i | i∈stmt^{-1}[{GetPresence}] ∧ iff(i)=N ∧ cP(v) ∈ sub(arg(i)) ∪ {arg(i)}
× {N+1}
end

3.5.2 The interpretation of generated events. The second event recognizes the generated instruction and performs the dynamic part of the refined event: the presence of the argument of the instruction is checked and the reaction is updated. Note that presence check could be removed: within the bloc of code skipped by the iff pointer, the presence condition is known to be satisfied. Due to lack of space, this optimization is not detailed here.

event smallStep_P **refines** smallStep_P
where
@pc pc < N // *end of generated code ¬reached*

```
    @arg stmt(pc) = Assign   // current statement is an Assign
    @p cP(arg(pc)) ∈ dom(ps)  // the clock of the assigned variable is present
  with
    @v v = arg(pc) // the processed variable is the argument of the statement
  then
    @ps ps(arg(pc)) := eqns(arg(pc))( dep(arg(pc)) ◃ ps)
    @npc pc := pc + 1
end
```

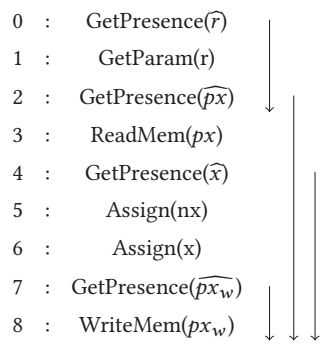
Absence being managed through the update of the `iff` pointers during the generation of all small step statement related to presence, only remains the dynamic part which consists in skipping a block of instructions in case a presence test fails. The new event declares the values of the parameters of the refined event.

```
event smallStep_mA refines smallStep_mA
  when
    @pcN pc < N // end of generated code ¬reached
    @pc stmt(pc) = GetPresence
    @pcnz pc > 0 // it is ¬the first statement
    @abs FALSE ∈ isPresent(arg(pc)↦pdep(arg(pc)) ◃ ps) // absence allowed
  with
    @v v = arg(pc)   // the processed variable
    @V V = arg[pc. .iff(pc)−1] // the set of variables to be skipped
  then
    @npc pc := iff(pc) // jump to next block
end
```

At last, the big step event is modified to take into account the new state variables: the compilation should have been performed before, the small steps should have been executed. It has to be noted that, except for this last event, compilation and execution of generated instructions can be interleaved during the computation of the first reaction of the run. This property is partially lost if presence test optimization is considered: if execution jumps to the end of the code as a consequence of a presence test failure, the compiler could insert an optimized statement (without presence check) at this point.

```
event bigStep refines bigStep
  where
    @g ran(arg) = 𝒱 // compilation ended
    @pc pc = N // end of small steps
  then
    @ps ps := ∅
    @d pc := 0
end
```

3.5.3 Example. The following table is an example of sequence of statements that could be obtained by executing the convergent events according to their ordering constraints. The table contains the sequence of statements (`stmt`) together with their arguments (`arg`). The arrows represent the `iff` jumps when the source condition is false. We remark that our specification of the compilation phase is actually non deterministic, other sequences of code could be generated.



3.6 Development overview

All the preceding development has been performed under the Rodin framework for Event-B. For all the development (contexts and machines), the tool generated 17 proof obligations for contexts and 239 proof obligations for machines. Most of the proof obligations could be discharged automatically thanks to the SMT solvers (CVC3-4, Z3,veriT). Some proof obligations needed manual quantification instantiation and case analysis.

We remark that unlike usual assistant theorem provers [9, 14, 15], elementary proof obligations are generated automatically, e.g., the preservation of each conjunct of the invariant by each action of each event. However, concerning Rodin, strategy languages are still missing.

4 RELATED WORK

In this section we review some related works. In the following, we position our work with respect to them.

4.1 Clock calculus

Here, we have adopted an axiomatic approach: in the context `cMeta_SUB`, we have specified a subsumption relation compatible with data dependency. The work of [3] can be seen as a concretization of this specification. As a future work, we intend to validate some of these constructions by taking into account the language to which they apply.

4.2 Translation validation

Translation validation has been pioneered by A. Pnueli et al. [16]. Recently, [13] have pushed forward this approach for the Signal language. The basic idea is that *each individual translation* is followed by a validation phase which verifies that the produced code *implements* the source code. Unlike this approach, here we are interested in the compilation approach: which verifies the compiler once for all. It follows that any produced code is guaranteed to be correct.

4.3 Compcert

The CompCert project [11] aims at the development of a formally verified C compiler. This work is far more complex than our one and could be used as a back end of a synchronous language to C compiler. However, one point is interesting to be discussed and is put forward in the CompCert project. It is related to what means

for a compiler to be correct? It should preserve the semantics of the source code, i.e., preserve some observable behaviors, which is ultimately expressed as proving a *forward simulation for safe programs* property. In this paper, a safe program is a program that can be put in solved form. Forward simulation is a consequence of refinement. Observable behaviors are sequences of reactions. Thus, Event-B brings us a methodology to incrementally build a similar proof (for a much less complex language).

4.4 Constructive semantics

A so called constructive semantics has been proposed for various synchronous languages [21]. This semantics aims at specifying how to compute output variables from inputs for each reaction. As in our case, small steps are proposed to solve constraints upon presence and values of the variables of the program. Specific rules are provided for various synchronous languages. The main difficulty of this approach, which is linked to the considered languages, is to find a trigger, i.e. a signal of which presence can be enforced in order to determine the presence and value of all other signals without failing to find a global solution when solving incrementally the constraint system. The authors define several classes of processes (reactive, deterministic, isochronous, ...) ensuring convergence properties. In this paper, we do not address these aspects and suppose that constraints have been put in solved form, which guarantees the ability to incrementally build a solution and has the same consequences as isochrony[21], i.e., the ability to non deterministically select a non empty subset of triggers to be present. Then, we mainly focus on refinement-based methodology to extract an optimizing compiler and the interpretor of the compiled code from the small step semantics which is itself shown to conform with the big step semantics. Furthermore, our framework is independent from a specific synchronous language.

5 CONCLUSION

In this paper, we have presented the development of a generic compiler for synchronous languages. Our main contribution was to structure such a development as successive refinement steps. Through these steps, we have introduced progressively the main ingredients of such a compilation process: presence of variables, clock calculus, optimization and code generation. Properties preservation was ensured thanks to the semantics of refinement supported by the Event-B formal method. With respect to the expression of the properties, set theory has revealed well suited to express: as well atomicity in high level specifications thanks to the composition of relations, as well the progressive approach thanks to partial functions. The proof obligations have been generated by the Rodin platform. All the proofs have been done. These proofs were made easier by the automatic SMT provers.

We envision to carry on this work with respect to two directions: first we would like to instantiate this generic compiler by well known synchronous languages as [5, 20]. Second, also, we would like to adapt code generation for targeting concurrent architectures [2].

REFERENCES

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press. http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521895569

[2] Joaquin Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. 2015. Denotational fixed-point semantics for constructive scheduling of synchronous concurrency. *Acta Inf.* 52, 4-5 (2015), 393–442. https://doi.org/10.1007/s00236-015-0238-x

[3] Tochéou Pascalin Amagbegnon, Loïc Besnard, and Paul Le Guernic. 1994. *Arborescent canonical form of boolean expressions.* Research Report RR-2290. INRIA. https://hal.inria.fr/inria-00074382

[4] André Arnold. 1994. *Finite transition systems - semantics of communicating systems.* Prentice Hall.

[5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

[6] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*, Krisztián Flautner and John Regehr (Eds.). ACM, 121–130. https://doi.org/10.1145/1375657.1375674

[7] John Derrick and Graeme Smith. 2012. Temporal-logic property preservation under Z refinement. *Formal Aspects of Computing* 24, 3 (2012), 393–416. https://doi.org/10.1007/s00165-011-0177-4

[8] Abdoulaye Gamatié. 2010. *Designing Embedded Systems with the SIGNAL Programming Language - Synchronous, Reactive Specification.* Springer. https://doi.org/10.1007/978-1-4419-0941-1

[9] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. 1997. *The Coq Proof Assistant : A Tutorial : Version 6.1.* Research Report RT-0204. INRIA. 44 pages. https://hal.inria.fr/inria-00069967 Projet COQ.

[10] Nassima Izerrouken, Marc Pantel, and Xavier Thirioux. 2009. Machine Checked Sequencer for Critical Embedded Code Generator. In *International Conference on Formal Engineering Methods (ICFEM), Rio de Janeiro, Brazil.* Springer-Verlag.

[11] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363. https://doi.org/10.1007/s10817-009-9155-4

[12] R. Milner. 1989. *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[13] Van Chan Ngo, Jean-Pierre Talpin, and Thierry Gautier. 2015. Translation Validation for Synchronous Data-Flow Specification in the SIGNAL Compiler. In *FORTE , Grenoble, France, June 2-4, 2015 (LNCS)*, Susanne Graf and Mahesh Viswanathan (Eds.), Vol. 9039. Springer, 66–80. https://doi.org/10.1007/978-3-319-19195-9_5

[14] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg.

[15] S. Owre, J. M. Rushby, and N. Shankar. 1992. PVS: A Prototype Verification System. In *11th International Conference on Automated Deduction (LNAI)*, Deepak Kapur (Ed.), Vol. 607. Springer-Verlag, Saratoga, NY, 748–752.

[16] Amir Pnueli, Ofer Strichman, and Michael Siegel. 1999. Translation Validation: From SIGNAL to C. In *Correct System Design, Recent Insight and Advances (LNCS)*, Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.), Vol. 1710. Springer, 231–255. https://doi.org/10.1007/3-540-48092-7_11

[17] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. 2006. Concurrency in Synchronous Systems. *Formal Methods in System Design* 28, 2 (2006), 111–130. https://doi.org/10.1007/s10703-006-7844-8

[18] Rodin [n. d.]. http://www.event-b.org/. ([n. d.]).

[19] Klaus Schneider. 2002. Proving the Equivalence of Microstep and Macrostep Semantics. In *Theorem Proving in Higher Order Logics, 15th Int. Conference, Hampton, VA, USA, August 20-23 (LNCS)*, Victor Carreño, César A. Muñoz, and Sofiène Tahar (Eds.), Vol. 2410. Springer, 314–331. https://doi.org/10.1007/3-540-45685-6_21

[20] Klaus Schneider, Jens Brandt, and Tobias Schuele. 2006. A Verified Compiler for Synchronous Programs with Local Declarations. *Electron. Notes Theor. Comput. Sci.* 153, 4 (June 2006), 71–97. https://doi.org/10.1016/j.entcs.2006.02.028

[21] Jean-Pierre Talpin, Jens Brandt, Mike Gemünde, Klaus Schneider, and Sandeep Shukla. 2014. Constructive Polychronous Systems. *Sci. Comput. Program.* 96, P3 (Dec. 2014), 377–394. https://doi.org/10.1016/j.scico.2014.04.009